HDL-based Synthesis of Reversible Circuits

A Scalable Design Approach

by
Zaid Saleem Ali Al-Wardi

A Dissertation Submitted in Partial
Fulfillment of the Requirements for
the Degree of Doctor of Engineering
- Dr.-Ing. -

University of Bremen, Faculty 3
Department of Mathematics and Computer Science

2018

Supervisor:
Prof. Dr Rolf Drechsler (University of Bremen, Germany)
Second Referee:
Prof. Dr Robert Wille (Johannes Kepler University Linz, Austria)

Date of the doctoral colloquium: April 23, 2018

To the memory of my father,

Prof. Dr Saleem Al-Wardi,

who left our world two years ago.

Acknowledgements

I would like to thank my advisor Prof. Dr Rolf Drechsler for providing me with the opportunity to complete my PhD thesis at the University of Bremen. I would like to thank him for encouraging my research and for allowing me to mature as a researcher, and I will always feel privileged having worked under his supervision for my PhD.

I have very special gratitude for Prof. Dr Robert Wille who has been truly a dedicated mentor. He has been actively interested in my work and has always been available to guide and help me. I highly appreciate his time, his tremendous academic support, and his encouragement to my contributions.

I am grateful to Prof. Dr Essam Abdul-baki and Dr Adheed Sallomi from Al-Mustansiriyah University in Baghdad-Iraq, for their unfailing support in obtaining the scholarship.

My most significant gratitude goes to my safety net, my family, who gave me all kinds of support during these years.

I would also like to acknowledge the friendly environment among all members of the computer architecture group, which had a positive impact on my personal and professional experience.

Finally, I would like to thank my committee members and my external examiner.

Thanks for all your encouragement!

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Due to continues developments in semiconductor technologies, more powerful computing devices are introduced to consumers every year. The well-known *Moore's Law*, published in 1965, predicted that the number of transistors, in circuits will be doubled each 24 months [1]. The period is often quoted as "18 months" due to Intel's executive David House, who adjusted the range for doubling the chip performance due to combination of effects from other factors in addition to the number of transistors.

Despite the original prediction claimed, in the original paper that it would only be valid for one decade (i.e., until 1975), Moore's continues to be valid today. This long-term expansion encourages consumers to be more demanding for faster, smaller, more functional and less energy consuming computing machines, which leads developers to be more competitive to produce new applications by harnessing any cutting edge technology as soon as it becomes available. Unfortunately, this gluttony towards more powerful computing machines is expected to be challenged by physical boundaries in the near future. In other words, alternative computational paradigms must be established soon to avoid a potential bottleneck.

A promising alternative is based on reversible computation [2], which exclusively allows bijective computations. In circuits based on reversible logic operations, all computations can be reverted, i.e., conducted with no information loss. On the other hand, conventional computation is based mostly on irreversible operations, e.g., a simple AND operation is irreversible, since knowing the output of an AND gate is not enough to know the values of its inputs. This suggests an entirely new computational paradigm. A circuit design flow for reversible systems is evolving during recent years with methodologies for reversible circuit description, synthesis [3, 4, 5], optimisation [6], simulation [7], verification and validation [8]. These methodologies are considered elementary as compared to the elaborated conventional design flow, which emerged over the last three decades and is supported by a wide-range of powerful tools on each level of abstraction. Consequently, significant contributions and efforts are still expected before reversible computing is accepted as a practical alternative to conventional computing. Despite this, reversible computation is already recognised as being relevant to some interesting applications, such as:

- **Low power computation** may significantly profit from reversible circuits in the future. This is due to observations by Landauer who states that power is always dissipated when information is lost during computations independent of the applied technology [9, 10]. Hence, all computing machines following the conventional paradigm always lose power if irreversible operations are performed (including a simple AND operation, for example). Although the fraction of power lost is negligible today, it will become substantial with continued miniaturisation. Since reversible computations are information loss-less (i.e., inputs can always be restored from the outputs and vice versa), the power loss can significantly be reduced or avoided with this alternative paradigm [2, 11].

- **Adiabatic circuits** utilises signals that switch their states very slowly to avoid power losses [12]. When the power dissipation from switching transitions is suppressed to a minimum, the static power dissipation caused by leaking devices in advanced, extremely miniaturised process technologies will become substantial. Regardless of the computing paradigm, static energy is present in virtually all transistor circuits. However, reversible circuits have the advantage that they naturally are suited for adiabatic switching without the need for extra circuitry.

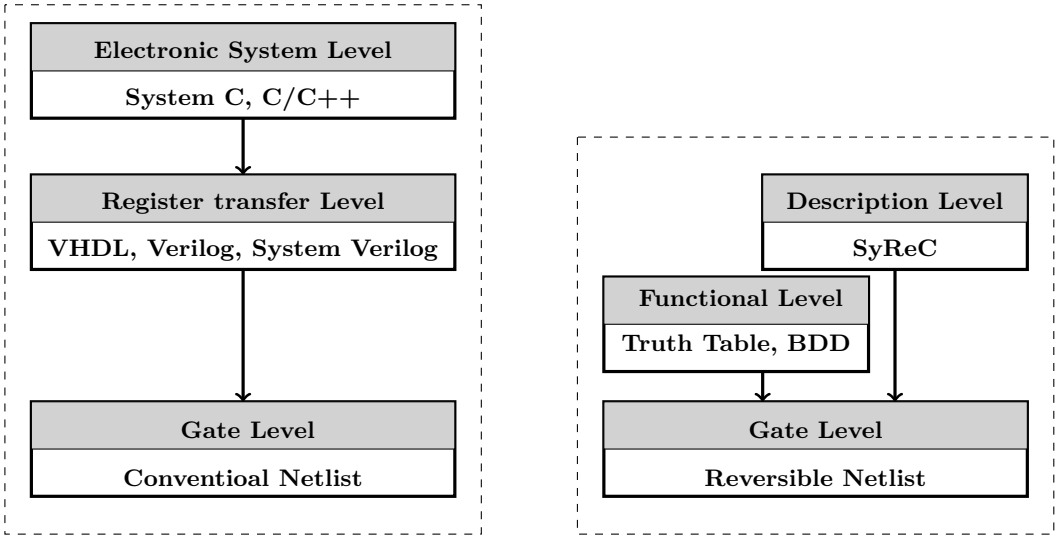
- **Encoding and decoding devices** realise reversible one-to-one mapping and, consequently, allow for a reversible computing paradigm. However, so far, most of these devices are implemented in a conventional (i.e., irreversible) manner and miss potential benefits in their design. An obvious application for encoders and decoders is in multimedia domains. Moreover, on-chip interconnections are increasingly making use of encoders and decoders to modify the communication between components of a system-on-chip device [13].

- **Quantum computation** offers the promise of more efficient computing for problems that are of exponential difficulty for conventional computing [14]. Considering that many of the established quantum algorithms include a significant Boolean component, it is crucial to have efficient methods to synthesise quantum gate realisations of Boolean functions. Since any quantum operation is inherently reversible, reversible circuits can be exploited for this purpose.

- **Program inversion** addresses how to derive the inverse of a given program automatically. As most existing programs follow the conventional (i.e., irreversible) computation paradigm, program analysis techniques [15] or interpretive solutions [16] are applied so far. However, programs based on reversible computation would allow an inherent and obvious program inversion.

1.1 Scalable Design Flow for Reversible Circuits

A design flow is a set of procedures guiding designers to progress from a specification for a chip to its final chip implementation in an error-free way [17]. The design of conventional circuits, from a structural perspective, is a hierarchical flow composed of several abstraction levels, including an electronic system, register transfer, gate and transistor levels. Many design tools have been developed and are accepted by designers, such as modelling languages, system description languages, and hardware description languages [18] (see Figure 1.1(a)). Furthermore, these design methodologies are supported by various powerful approaches for simulation, verification, validation, and debugging to ensure the correctness of a designed circuit or system [19].



(a) Established flow for conventional circuits.

(b) Existing flow for reversible circuits.

Figure 1.1: Design flows for conventional and reversible circuits.

Logic synthesis fits between the register transfer and gate levels. This flow represents the process of mapping *Hardware Description Language (HDL)* specifications into netlists of gate specifications. The existing design methodologies for reversible circuit synthesis remain far from modern industrial needs. In fact, although researchers have considered the basic tasks of synthesis, verification, and debugging, tools for reversible circuit design are still applied on a small scale [20] (see Figure 1.1(b)).

Hardware description languages, such as *VHDL*, *Verilog* and *System Verilog*, are used for the tasks of the register transfer level of conventional design flow, such as describing, simulating, synthesising, and verifying the desired systems [21, 22, 23]. They became a corner-stone in the conventional design flow because they are practical, powerful, scalable, and easy-to-use design tools.

1.1.1 State of the Art

A dedicated reversible hardware description language, SyReC, has been introduced [24]. It allows for the specification and automatic synthesis of reversible circuits. Experiments show that SyReC is capable of handling large designs beyond the capacity of other design approaches (e.g., a reversible RISC CPU [25]).

Nevertheless, it is known that minimum circuits are not guaranteed using this approach. This is a serious drawback in reversible circuits where circuit resources are limited, especially when it comes to quantum technology. Consequently, the HDL approach will be considered in practice only when it achieves better quality circuits.

1.1.2 Thesis Contribution

In this dissertation, we analyse the HDL design of reversible circuits along with weaknesses and potentials of this approach. Modifications on different stages of HDL-based design flow are proposed in this dissertation to optimize the synthesis of reversible circuits with better quality. The ideas presented here are based on the following peer-reviewed papers along with additional unpublished work:

1. [26] **Towards Line-aware Realisations of Expressions for HDL-based Synthesis of Reversible Circuits**
   Zaid Al-Wardi, Robert Wille, Rolf Drechsler
   Reversible Computation **7** (2015), Springer, LNCS 9138, pp. 233–247.

2. [27] **Rewriting HDL Descriptions for Line-aware Synthesis of Reversible Circuits**
   Zaid Al-Wardi, Robert Wille, Rolf Drechsler
   International Symposium on Multiple-Valued Logic **46** (2016), IEEE, pp. 39–53.

3. **Optimized Realizations of Expressions for HDL-based Synthesis of Reversible Logic Circuits**
   Zaid Al-Wardi, Robert Wille, Rolf Drechsler
   International Workshop on Post-Binary ULSI Systems **25** (2016), poster presentation.

4. [28] **Extensions to the Reversible Hardware Description Language SyReC**
   Zaid Al-Wardi, Robert Wille, Rolf Drechsler
   International Symposium on Multiple-Valued Logic **47** (2017), IEEE, pp. 185–190.

5. [29] **Towards VHDL-based Design of Reversible Circuits**
   Zaid Al-Wardi, Robert Wille, Rolf Drechsler
   Reversible Computation **9** (2017), Springer, LNCS 10301, pp. 102–108.

6. [30] **Synthesis of Reversible Circuits Using Conventional Hardware Description Languages**
   Zaid Al-Wardi, Robert Wille, Rolf Drechsler
   International Symposium on Multiple-Valued Logic **48** (2018), IEEE, (accepted).

1.2 Thesis Outline

The dissertation is composed of the following chapters:

Chapter 2 provides the required background the review of definitions and notations for Boolean functions and reversible logic gates. Reversible circuits are reviewed with metrics used to evaluate quality as well as brief overview of the categories of circuit synthesis methods. The pros and cons of each category are also reviewed in this chapter.

Chapter 3 previews the dedicated reversible language SyReC, with details. The synthesis scheme of the SyReC Specifications and possible optimised realizations are also discussed.

Chapter 4 proposes a set of rules to optimise SyReC programs, such that it realise desired descriptions in circuits with fewer lines or less cost. As a result a specific programming style is suggested.

Chapter 5 introduces line-aware realisations of HDL expressions. Less constant inputs are exploited to realize complex expressions, which combine many operations. The chapter proposes more than one scenario to compute the same expression.

Chapter 6 considers the conventional hardware description language, VHDL, as an alternative to realise reversible circuits. Realisations of VHDL statements are investigated, and the restrictions associated with reversible circuit paradigm are considered.

Chapter 7 investigates some possible enhancements on SyReC grammar to simplify descriptions, including simple control logic statements and data operations. Moreover, the suggested grammar enables the language to accept some parts to be replaced by circuits realised with other synthesis methodologies.

Chapter 8 provides a final summary and conclusions for the thesis.

Chapter 2

Reversible Computation

This chapter introduces the basics of reversible logic and other concepts to offer a complete background required for the dissertation. The first section overviews the basic definitions, and notations, and different ways to describe Boolean functions. The second section provides a summary of the principles of reversible logic, definitions of reversible gates and circuits, and metrics defined to evaluate the quality of these gates and circuits. The third section reviews basic methodologies for reversible circuit synthesis as well as approaches to optimise these circuits.

2.1 Boolean Functions

Boolean computations can be defined as functions over Boolean values $\mathbb{B} \in \{0, 1\}$, or more precisely:

Definition 1. *A* Boolean function f *is a mapping* $f\colon \mathbb{B}^n \mapsto \mathbb{B}^m$ *with* n *inputs and* m *outputs.*

There are $2^{(m \cdot 2^n)}$ possible Boolean functions in a system with n-inputs and m-outputs. Elementary functions, such as *conjunction* ($\wedge$,AND), *disjunction* ($\vee$,OR), *negation* ($\neg$,NOT), and *inequality* ($\oplus$,XOR) are combined in Boolean expressions to describe complex functions fully.

The most straightforward and direct way to specify the behaviour of a Boolean function is by enumerating the output's response to each input bit-vector. This input/output relationship is commonly enumerated in a tabular form, called a *truth table*, in which all possible input bit-vectors are listed on the left side of the table, and the corresponding output bit-vectors computed by the function are listed on the right side of the table. For instance, Figure 2.1 shows truths tables for the basic logical operations. The number of rows in a truth table equals the number of possible input bit combinations, 2^n, where n is the number of inputs. This exponential growth in the sizes of truth tables limits these methodologies in a practical sense to functions with only a small number of inputs.

x_1	x_2	y
0	0	0
0	1	0
1	0	0
1	1	1

(a) AND ($x_1 \wedge x_2$)

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	1

(b) OR ($x_1 \vee x_2$)

x	y
0	1
1	0

(c) NOT $\overline{x}$

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	0

(d) XOR ($x_1 \oplus x_2$)

Figure 2.1: Basic Boolean functions.

Alternative compact representations are practical for describing Boolean functions with a higher number of inputs, such as simplified *Boolean equations* [31] or graphical (e.g., *Binary Decision Diagrams* BDDs [32]). These alternative representations can have short descriptions for functions, but also show exponential growth in the worst case.

Modern designs deal with a substantial number of signals rendering truth tables useless at this scale. Here, other approaches are used, such as *Hardware Description Languages* (HDLs), that are introduced to facilitate high-level descriptions that can scale well [18].

2.2 Reversible Logic

Reversible computations are restricted to bijective operations, which have inverses. In other words, we can inversely map any output to the corresponding input. In this section, the basics of reversible logic are introduced. First, we present the properties of reversible functions and embedding irreversible functions. The second part defines common reversible gates that are used throughout this work. Finally, we introduce reversible circuits and their characteristics and metrics as well as basic synthesis approaches.

2.2.1 Reversible Functions

Definition 2. *A function* $f\colon \mathbb{B}^l \mapsto \mathbb{B}^l$ *is called* reversible *if it is bijective, i.e., if each input-bits pattern is uniquely mapped to a corresponding output-bits pattern and vice versa. Otherwise, it is called* irreversible.

f is reversible, if and only if:

1. the number of inputs equals the number of outputs $= l$.

2. each output-bits pattern appears only once in the truth table.

x_1	x_2	x_3	y_1	y_2	y_3
0	0	0	0	0	0
0	0	1	1	0	0
0	1	0	0	1	0
0	1	1	**0**	**1**	**1**
1	0	0	**0**	**1**	**1**
1	0	1	1	0	1
1	1	0	1	1	1
1	1	1	1	1	0

(a) f_1 : Irreversible function

x_1	x_2	x_3	y_1	y_2	y_3
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	1	0	0
0	1	1	1	0	1
1	0	0	0	0	1
1	0	1	0	1	1
1	1	0	1	1	0
1	1	1	1	1	1

(b) f_2 : Reversible function

Figure 2.2: Examples of Boolean functions.

Example 1. *Figure 2.1(c) shows the truth table for a NOT operation. The function is reversible (bijective) because the number of inputs* $(n = m = 1)$ *and each input-bits pattern is uniquely mapped into an output-bits pattern. On the other hand, AND, OR, and XOR functions (as described in Figure 2.1(a), 2.1(b), and 2.1(d)) are non-reversible because they have inputs* $(n = 2)$ *and outputs* $(m = 1)$*, i.e.,* $(n \neq m)$*. The function* f_1 *presented in Figure 2.2(a) is also irreversible because the inputs are not uniquely mapped into the outputs (e.g., both inputs* 011 *and* 100 *map to the same output* 011*). In contrast, the function* f_2 *outlined in Figure 2.2(b) is reversible, since each input vector maps to a unique output vector and the number of inputs is equal to the number of outputs* $(n = m = 3)$*.*

2.2.2 Embedding of Irreversible Functions

An irreversible function can be embedded into a reversible specification by adding extra variables to achieve a bijective function. An embedding is not unique, and the choice of embedding can have a very significant effect on the number of the variables of the resulting function [33, 34].

Definition 3. *A reversible function* $g\colon \mathbb{B}^{(n+p)} \mapsto \mathbb{B}^{(m+k)}$ embeds *the irreversible* $f\colon \mathbb{B}^n \mapsto \mathbb{B}^m$*, if* $f_i(X) = g_i(X)$ *for each* $X \in \mathbb{B}^n$ *and each* $i \in \{1, 2, \ldots, m\}$*. The function g is called an* embedding *and the additional k outputs of g are referred to as* **garbage outputs***. Furthermore, p additional input variables are added such that* $(n+p) = (m+k) = l$ *to obtain a reversible function for the embedding g. The additional p inputs are referred to as* **constant inputs***.*

More precisely, given an m-output irreversible function f on n variables, a *reversible* function g with $m + k$ outputs is determined such that g agrees with f on the first m components. Then, bijectivity can readily be achieved, e.g., by adding p additional inputs such that f evaluates to its original values in case these inputs are assigned the constant

C_{in}	x_2	x_1	C_{out}	S
0	0	0	0	0
0	0	1	**0**	**1**
0	1	0	**0**	**1**
0	1	1	1	0
1	0	0	**0**	**1**
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

(a) 1-bit full adder $f\colon \mathbb{B}^3 \mapsto \mathbb{B}^2$

C	C_{in}	x_2	x_1	C_{out}	S	γ_1	γ_2
0	**0**	**0**	**0**	**0**	**0**	0	0
0	**0**	**0**	**1**	**0**	**1**	0	1
0	**0**	**1**	**0**	**0**	**1**	1	0
0	**0**	**1**	**1**	**1**	**0**	1	1
0	**1**	**0**	**0**	**0**	**1**	0	0
0	**1**	**0**	**1**	**1**	**0**	0	1
0	**1**	**1**	**0**	**1**	**0**	1	0
0	**1**	**1**	**1**	**1**	**1**	1	1
1	0	0	0	1	0	0	0
1	0	0	1	1	1	0	1
1	0	1	0	1	1	1	0
1	0	1	1	0	0	1	1
1	1	0	0	1	1	0	0
1	1	0	1	0	0	0	1
1	1	1	0	0	0	1	0
1	1	1	1	0	1	1	1

(b) 1-bit full adder embedded in $g\colon \mathbb{B}^4 \mapsto \mathbb{B}^4$

Figure 2.3: Embedding of the 1-bit full adder function.

value 0 and each output pattern that is not in the image of g is arbitrarily distributed among the new input patterns.

Let $\mu(f)$ denote the number of occurrences of the most frequent output pattern in the truth table of a. Then, $\kappa(f) \stackrel{\text{def}}{=} \lceil \log_2 \mu(f) \rceil$ is the minimum number of garbage outputs (denoted by k) required to convert an irreversible function to a reversible function. Thus, if the number of garbage-bits $k = \kappa(f)$, then the embedding g is called *optimal*. The worst case $\mu(f) = 2^n$ (the total number of rows in the truth table) leads to $\kappa(f) = \lceil \log_2(2^n) \rceil = n$, i.e., n garbage bits are sufficient, even in the worst case, to embed any random binary function [35]. In other words, there exists a reversible function $g\colon \mathbb{B}^{(n+m)} \mapsto \mathbb{B}^{(n+m)}$ to embed a random Boolean function $f\colon \mathbb{B}^n \mapsto \mathbb{B}^m$. Different algorithms that perform an embedding of irreversible functions based on their truth table description have been proposed in the past [33, 34, 36].

Example 2. *The* 1*-bit full adder function* $f\colon \mathbb{B}^3 \mapsto \mathbb{B}^2$ *specified in Figure 2.3(a) is obviously an irreversible Boolean function. The most frequent output pattern in* f *is* 01*. This pattern is repeated three times with input patterns* 001*,* 010*, and* 100*. Then,* $\mu(f) = 3$*. As can be seen, the number of additional garbage outputs are* $k = \kappa(f) = \lceil \log_2 \mu(f) \rceil = 2$*, hence the embedding* g *is optimal. The function can be embedded into a reversible function,* $g\colon \mathbb{B}^4 \mapsto \mathbb{B}^4$*, as illustrated in Figure 2.3(b). A constant input* C *and two garbage outputs* (γ_1, γ_2) *are added to ensure the reversibility of the final embedded functional, i.e., the same number of inputs and outputs.*

(a) NOT (b) CNOT (c) MCT

Figure 2.4: Toffoli gates.

2.2.3 Reversible Gates

Reversible functions can be realised by reversible circuits in which each variable of the function is represented by a *circuit line* to maintain the bijectivity property of the reversible function. There exist different gate libraries used to build reversible circuits. However, in the scope of this work, we restrict ourselves to those most commonly used containing the *Toffoli gate* [37] and the *Fredkin gate* [38]. For this purpose, each gate g_i in the circuit is denoted by $g(C, T)$ with *control lines* $C \subset X$ and *target lines* $T \subseteq X \setminus C$.

2.2.3.1 Toffoli Gates

A Toffoli gate has one target line $T = \{x_t\}$ and maps the input $X = \{x_1, x_2, \ldots, x_n\}$:

$$(x_1, \ldots, x_n) \mapsto (x_1, \ldots, x_{t-1}, x_t \oplus \bigwedge_{x_j \in C} x_j, x_{t+1}, \ldots, x_n),$$

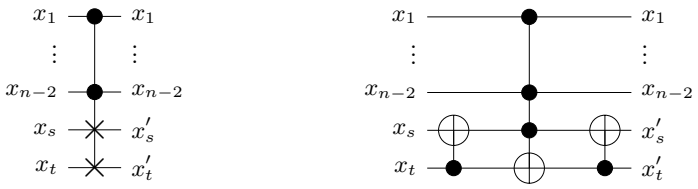
i.e., the value on line x_t is inverted if and only if all control values are assigned 1. The Toffoli gate is called a **NOT** if $|C| = \emptyset$. Here, the gate maps the single input, such that $x \mapsto (x \oplus 1)$, i.e., output is unconditionally inverted. On the other hand, a Toffoli gate is called a **CNOT**, or *Feynman* gate, if it has one control line x_c and one target line x_t. Here, the gate maps $(x_c,\ x_t)x \mapsto (x_c,\ x_t \oplus x_c)$, while Toffoli gates with more than one control line are usually referred to as *Multiple control Toffoli*, **MCT**, as seen in Figure 2.4. For a graphical representation of Toffoli gates, solid black circles (•) indicate controls and target lines are denoted with the symbol $\oplus$.

2.2.3.2 Fredkin Gates

A Fredkin gate has two target lines $T = \{x_s, x_t\}$ and maps the input

$$(x_1, \ldots, x_n) \mapsto (x_1, \ldots, x_{s-1}, x'_s, x_{s+1}, \ldots, x_{t-1}, x'_t, x_{t+1}, \ldots, x_n),$$

with $x'_s = (\bar{\xi} x_s \oplus \xi x_t)$, $x'_t = (\bar{\xi} x_t \oplus \xi x_s)$, and $(\xi = \bigwedge_{x_j \in C} x_j)$, i.e., the values of the target lines are interchanged (swapped) if and only if all control values are assigned 1. A Fredkin gate is referred to as a SWAP gate if $|C| = \emptyset$. For a graphical representation of Fredkin gates, solid black circles (•) indicate controls and target lines are denoted with the symbol $\times$, as seen in Figure 2.5(a). Any Fredkin gate can be realised by cascading three Toffoli, as in Figure 2.5(b).



(a) Symbol (b) Toffoli implemetation

Figure 2.5: Fredkin gate.

2.2.4 Reversible Circuits

A reversible circuit is an acyclic combinational logic circuit in which all gates are reversible and interconnected without explicit fan-outs and loops [39]. Therefore, reversible circuits can be built as a cascade of reversible gates $G = g_1 \dots g_d$. Each gate g_i realises a reversible function $f_i : \mathbb{B}^l \mapsto \mathbb{B}^l$. The function realised by the circuit is the composition of the functions realised by the gates, i.e., $f = f_1 \circ f_2 \circ \dots \circ f_d$. In this circuit paradigm, fan-out and feedback are not directly allowed.

Example 3. *Figure 2.6 shows a reversible circuit combined using Toffoli gates. Three primary inputs $X = (x_1, x_2, x_3)$ are mapped to two valid outputs $Y = (y_1, y_2)$. A constant input is applied to the circuit. On the other hand, the circuit has two garbage outputs.*

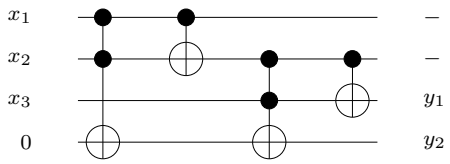


Figure 2.6: An example reversible circuit.

By definition, reversible computations are bijective, i.e., for each reversible circuit, $G = g_1\ g_2\ \dots\ g_d$ there exists an inverse circuit G^{-1}, such that the cascade $G\ G^{-1} = I$, where I is a unity mapping $I\colon X \mapsto X$. Toffoli and Fredkin gates are self-inverse gates, i.e., $g_i = g_i^{-1}$. Consequently, by backward arranging the gates backwards, the inverse circuit is computed $G^{-1} = g_d\ g_{d-1}\ \dots\ g_1$. Hence, for $G = G_1\ G_2$ the inverse $G^{-1} = G_2^{-1}\ G_1^{-1}$.

Irreversible Boolean functions can be synthesised to a reversible circuit after embedding them to reversible functions. Therefore, in general, a reversible circuit contains l lines with n primary inputs and p constant inputs with $(n + p) = l$. At the output side, there are m primary outputs and k garbage outputs with $(k + m) = l$. Figure 2.7 depicts the general structure of a reversible circuit inspired from [39]. Note that when the function f is bijective, there are neither constant inputs nor garbage outputs. This follows from [3] where it was shown that any reversible function $f\colon \mathbb{B}^n \mapsto \mathbb{B}^n$ could be realised by a reversible circuit with n lines when using MCT gates. This means that it is not necessary to apply any constant input to realise the circuit.

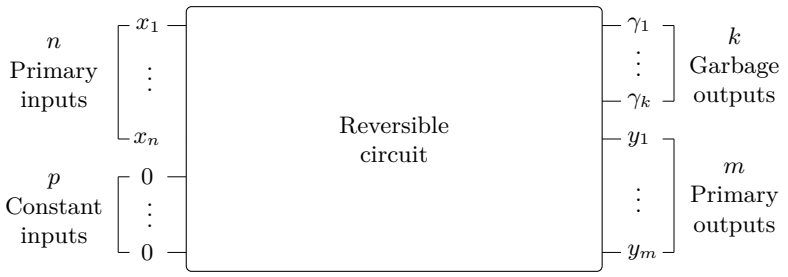


Figure 2.7: Reversible circuit structure.

Example 4. *Figure 2.8(a) shows a truth table for an embedded AND function* $(x_1 \wedge x_2)$ *with a constant input* c *and two garbage outputs* γ_1 *and* γ_2*. On the other hand, Figure 2.8(b) shows an embedded XOR function* $(x_1 \oplus x_2)$ *where only one garbage* γ *is required. Here, no constant input is needed to compute this function (reversible computation).*

c	x_1	x_2	y	γ_1	γ_2
0	**0**	**0**	**0**	0	0
0	**0**	**1**	**0**	0	1
0	**1**	**0**	**0**	1	0
0	**1**	**1**	**1**	1	1
1	0	0	1	0	0
1	0	1	1	0	1
1	1	0	1	1	0
1	1	1	0	1	1

(a) Embedded AND

x_1	x_2	y	γ
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	1

(b) Embedded XOR

Figure 2.8: Embedded functions.

2.2.5 Metrics of Reversible Circuits

It is important to evaluate the resulting circuits to compare different synthesis approaches. Depending on the target application, different metrics are applied to measure the quality of a given circuit.

2.2.5.1 Number of Lines

The number of lines l refers to the total number of the input or output variables used in a reversible circuit. If the function to be synthesised is reversible, then the number of circuit lines can be equal to the number of the inputs. However, in the case where the Boolean function to be synthesised is irreversible, then additional variables (i.e., constant inputs and garbage outputs) are unavoidable [36]. For example, the circuit in Figure 2.6 has a total number of lines $l = 4$, one of which has a constant input, which indicates some irreversible function embedded within the reversible circuit.

Good circuit realisations try to keep the number of lines as small as possible. This is motivated by the fact that, in the domain of quantum computations, each circuit line represents a quantum-bit (qubit), which is a very limited resource [40]. Therefore, several optimisation approaches that target a reduction of the number of lines in reversible circuits have been proposed [41].

2.2.5.2 Gate Count

The gate count d refers to the total number of gates in a reversible circuit, e.g., $d = 4$ in Figure 2.6. This metric is used to evaluate a given realisation for a reversible function. However, it is a poor measure of actual reversible circuit complexity because different gates can have dramatically different complexities depending on the technology used in the physical implementation of the circuit. Consequently, minimisation based on this metric does not guarantee a minimum complexity of the physical circuit. In fact, it is possible to reduce the overall cost using extra gates [42].

2.2.5.3 Technology-dependent Cost Metrics

The complexity of reversible circuits is measured by different cost metrics, which are more technology dependent, such as quantum costs and transistor costs. The cost C_G of a reversible circuit $G = g_1 g_2 \dots g_d$ is the summation of the costs of the individual gates. i.e., $C_G = \sum_{i=1}^{d} C_{g_i}$, where C_{g_i} is the cost of the individual gate g_i (the i^{th} in the circuit cascade).

While the transistor cost model estimates the cost of the circuit in terms of the number of CMOS transistors [43], the quantum cost model estimates the cost of the circuit in terms of the number of elementary quantum gates [14]. Both metrics define the cost of a single Toffoli/Fredkin gate depending on the number of control lines.

- **Transistor cost model:** The transistor cost of a gate estimates the effort needed to realise a reversible gate in CMOS according to [11] defined to be $C_g = (8 \times n)$, where n is the number of control lines in the gate [43]. Then, the transistor cost metric of a circuit is the total sum of transistor costs of all reversible gates in the circuit.

- **Quantum cost model:** The quantum cost of a Toffoli gate, as introduced in [44] and optimised in [45], is given in Table2.1, where n denotes the number of control lines for the gate and l denotes the total number of circuit lines. The quantum cost depends on the number n of control lines as well as the number $(l - (n + 1))$ of *empty lines* neither used as a control line nor a target line for the gate. These free-to-use lines are also called *ancilla* lines. More empty lines generally lead to cheaper gate realisation, although, for each gate size, there is a minimal cost which is not reduced by having further extra lines available, as outlined in Table 2.1a. On the other hand, the quantum cost of a Fredkin gate with n control lines is computed as the cost of a Toffoli gate of $n + 1$ controls plus the cost of two CNOT gates (see Table 2.1b).

Table 2.1: Quantum costs metrics of reversible gates.

a Multiple control Toffoli gates

$\#Control\ lines(n)$	$Transistor\ cost$	Quantum Cost		
		$ancella = 0$	$ancella = 1$	$ancella \geqslant n-2$
0	0	1	–	–
1	8	1	–	–
2	16	5	–	–
3	24	13	–	–
4	32	29	–	26
5	40	61	52	38
6	48	125	80	50
7	56	253	100	62
8	64	509	128	74
9	72	1021	152	86
$\geqslant 10$	$n \times 8$	$2^{n+1} - 3$	$24 \times (n+1) - 88$	$12 \times (n+1) - 34$

b Multiple control Fredkin gates

$\#Control\ lines(n)$	$Transistor\ cost$	Quantum Cost		
		$ancella = 0$	$ancella = 1$	$ancella \geqslant n-2$
0	0	3	–	–
1	8	7	–	–
2	16	15	–	–
3	24	31	–	–
4	32	63	54	40
5	40	127	82	52
6	48	255	102	64
7	56	511	130	76
8	64	1023	154	88
9	72	2047	178	100
$\geqslant 10$	$n \times 8$	$2^{n+2} - 1$	$24 \times (n+2) - 86$	$12 \times (n+2) - 32$

Example 5. *The reversible circuit in Figure 2.6 is a cascade of* $d = 4$ *Toffoli gates,* $g_1 \ldots g_4$*. Two (*g_1 *and* g_3*) have* $n = 2$ *control lines, i.e., each has a transistor cost of* 16 *and quantum cost of* 5 *(no ancella). The other two (*g_2 *and* g_4*) have* $n = 1$ *control line, i.e., each has a transistor cost of* 8 *and quantum cost of* 1*. In total, the circuit transistor cost metric is (*$16 \times 2 + 8 \times 2 = 48$*), while the quantum cost metric is (*$5 \times 2 + 1 \times 2 = 12$*).*

2.3 Reversible Circuit Synthesis

Synthesis is an essential phase in the design flow of reversible circuits. The goal is to synthesise a reversible circuit that computes the desired function efficiently, i.e., with fewer circuit lines and with lower total gate cost. Reversible circuit synthesis is approached from two directions, resulting in two main categories of synthesis, each with its pros and cons. The first category ensures a circuit with a minimal number of lines, as in [3,4,36,46,47,48,49,50]. These approaches are truth table-based, so they are limited by a small number of inputs. The second category includes hierarchical approaches that decompose a large function into smaller sub-functions, and these approaches make use of additional circuit lines to interconnect the sub-functions to realise the overall function [24, 51, 52]. In comparison, hierarchical approaches offer higher capacity in handling designs with a larger number of input signals, but they can not ensure circuits with minimal lines. Examples from both categories are introduced in the following.

2.3.1 Truth Table-based Synthesis

Truth table-based approaches realise functions as circuits with a minimum number of lines. Here, the function to be synthesised is embedded and described by a truth table (see Section 2.2.2). The number of lines required to compute the desired outputs is equal to the number of columns on one side of the truth table, i.e., either inputs or outputs ($l = (n + p) = (k + m)$). Here, lines are ensured to be at a minimum because the constant and garbage bits are also mathematically required for an optimal embedding of the desired function.

The *transformation-based synthesis* [47] is an example of this category. The idea is to traverse each row of a reversible truth table and match the output bit-vector with the input bit-vector starting from the least significant bit in the first row, until the most significant bit in the last row. When a mismatched bit is detected, the bit is altered for all rows that have $'1'$s in all bit positions that are $'1'$s in the mismatched row. This is what an MCT gate g_i does when its target line is positioned at the mismatched bit, and it has a control input at each line with value $'1'$ in this row. If the truth table rows are arranged in ascending order, then these gates do not alter bits in rows that were previously matched. When this matching is concluded, the circuit outputs match the inputs, i.e., an identity function is achieved. Hence, gates are appended starting from the output side of the circuit (g_d), until the last gate becomes the first from the input side (g_1), where d is the gate count equal to the number of steps required to match the outputs to the inputs totally. This synthesis approach results in circuits with a total number of lines n as defined above.

x_1	x_2	$x_1 \wedge x_2$	$x_1 \vee x_2$
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1

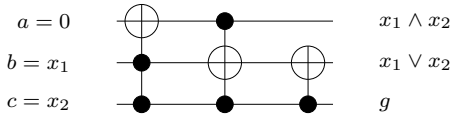
(a) Truth table of function f

0	x_1	x_2	$x_1 \wedge x_2$	$x_1 \vee x_2$	g
0	0	0	0	0	0
0	0	1	0	1	1
0	1	0	0	1	0
0	1	1	1	1	1
1	0	0	1	0	0
1	0	1	1	0	1
1	1	0	1	1	0
1	1	1	0	0	1

(b) Embedded f in a reversible truth table

(i)	*input* abc	*output* abc	*step*1 abc	*step*2 abc	*step*3 abc
0	000	000	000	000	000
1	001	0**1**$\underline{1}$	001	001	001
2	010	010	010	010	010
3	011	1**1**$\underline{1}$	$\underline{1}$**0**$\underline{1}$	**1**$\underline{11}$	011
4	100	100	100	100	100
5	101	1**0**$\underline{1}$	$\underline{1}$**1**$\underline{1}$	101	101
6	110	110	110	110	110
7	111	0**0**$\underline{1}$	011	**0**$\underline{11}$	111

(c) Transformation based appraoch



(d) Reversible circuit realisation

Figure 2.9: Transformation based synthesis of function f in Example 6.

Example 6. *Consider the* $\mathbb{B}^2 \mapsto \mathbb{B}^2$ *mapping* $f\colon (x, y) \mapsto (x \wedge y\ ,\ x \vee y)$*, which is an irreversible function (see truth table in Figure 2.9(a)). To synthesise a reversible circuit that computes* f*, first, the function should be embedded within a reversible mapping, which requires a third bit to be introduced in the truth table with a constant input value (see Figure 2.9(b)). Second, the transformation-based synthesis can be applied requiring three steps (i.e., MCT gates) to match all the output vectors to the corresponding input vectors (see Figure 2.9(c)). The resulting circuit is synthesised using three gates and within three lines (see Figure 2.9(d)).*

1. *The rightmost gate* g_3 *has its target at line* $T_3 = \{b\}$ *and a control at line* $|C_3| = \{c\}$*. This matches output column* b*, raw* 1 *with the corresponding input bit.*

2. *The middle gate* g_2 *has its target at line* $T_2 = \{b\}$ *and two controls at lines* $|C_2| = \{a, c\}$*. This matches step*1 *column* b*, raw* 3 *with the corresponding input bit.*

3. *The leftmost gate* g_1 *has its target at line* $T_1 = \{a\}$ *and two controls at lines* $|C_1| = \{b, c\}$*. This matches step*2 *column* b*, raw* 3 *with the corresponding input bit. The result is in step*3*, which is a full match to the input vectors.*

This approach requires the complete truth table to be available in memory, so due to an exponential growth in memory this approach practically bounds the capacity to relatively small number of inputs (up to 30).

2.3.2 Hierarchical Decomposition Synthesis

Hierarchical synthesis methods have been introduced as an alternative, which does not require explicit embedding, but do not guarantee a minimum number of circuit lines. In fact, the lines are often much more than the minimum [20].

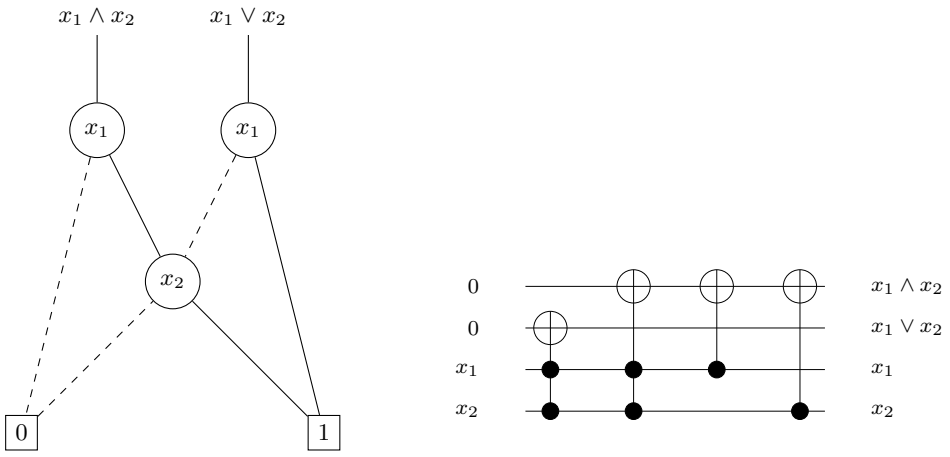
The general idea of hierarchical methods is to decompose the desired function f into a smaller set of functions (sub-functions). This decomposition is repeatedly applied until a circuit is realised for the sub-function or evaluated to a constant. Then, for each decomposition, the sub-circuit representing the respective operation can be synthesised. Finally, by composing all sub-circuits, a circuit representing the desired function is realised.

Shannon decomposition of an arbitrary function $f(x_1, \ldots, x_i, \ldots, x_n)$, is a well-known example of hierarchical decomposition, and is given by: $f = \overline{x_i} \cdot f_H \oplus x_i \cdot f_L$, where $f_L = f(x_1, \ldots, 0, \ldots, x_n)$ and $f_H = f(x_1, \ldots, 1, \ldots, x_n)$, i.e., the original function with input x_i is substituted by 0 and 1 in f_L and f_H, respectively. Here, $f : \mathbb{B}^n \mapsto \mathbb{B}^n$ is decomposed to $f_L : \mathbb{B}^{n-1} \mapsto \mathbb{B}^{n-1}$ and $f_H : \mathbb{B}^{n-1} \mapsto \mathbb{B}^{n-1}$, i.e., each has one input variable less than the original function f. In the same way, f_L and f_H can be decomposed into smaller sub-functions. This hierarchical decomposition reduces the problem size to moderate the exponential growth of functions.

Hierarchical function decomposition motivates compelling representations of Boolean functions, such as *Binary Decision Diagrams* (BDD), which are based on graph theory and provide efficient data-structures that can represent large functions more compactly than truth tables [32, 53]. BDD-based synthesis has been proposed for larger scale functions in the reversible circuit paradigm (up to 200 inputs) [52, 54, 55].

A node (a circle in the graph) represents a function or sub-function with an input-variable identifier written inside the node. Depending on the value of this input, it decides on either 1 referring to the continuous-line edge leads to the sub-function f_H, or 0 then the dashed-line edge leading to the sub-function f_L. This completes when it reaches a terminal node with constant value $\{0, 1\}$. BDD-based synthesis traverses this decision-diagram and substitutes each node with a predefined cascade of reversible gates [52].

Example 7. *Figure 2.10(a) shows a reduced BDD representation of the function described in the truth table in Figure 2.9(a). The reversible circuit realised from this approach is computed using a total of four lines, as shown in Figure 2.10(b), i.e., with one more line compared to the transformation-based approach. Here, two constant inputs are applied to the circuit instead of just one line.*



(a) BDD of f (b) Reversible circuit realisation

Figure 2.10: BDD-based synthesis of function f from Example 7.

Examples 6 and 7 describe a simple function. The more complex a function, the more constant input lines it exploits to be computed using hierarchical approaches, which is the primary drawback of this category. Also, obtaining a minimised BDD is not trivial.

As digital systems are rapidly becoming more complex, bit-level descriptions are no longer suitable to describe them. The hierarchical decomposition concept opens the door for more abstract descriptions, such as *Hardware Description Languages*, for complex systems in the conventional circuit paradigm. This high level description allows for modular and scalable designs [56, 57, 58]. Similarly, a dedicated reversible hardware description language *SyReC* is proposed to facilitate scalable synthesis of reversible circuits through common HDL means [24], and SyReC will be reviewed in Chapter 3.

2.3.3 Trade-off in Circuit Lines and Gate Costs

Here, many synthesis approaches are compared and their characteristics are observed. While minimum line methods result in circuits with fewer lines, often the respective quantum/transistor costs are higher in comparison to hierarchical synthesis. In contrast, hierarchical approaches need a significant number of additional constant inputs applied to circuits, but usually with lower quantum costs [20], which shows the complementary characteristics of these two categories. Figure 2.11 illustrates the problem with an abstract relation between the number of lines and the quantum cost to realise function f using different synthesis methods. Extreme results appear either with high quantum cost or with large number of lines. This motivates investigating possible trade-offs between circuit metrics to realise moderate circuits that are neither extreme in lines nor costs. A trade-off can be made in either direction, i.e., by adding lines to reduce the cost of a minimal lines circuit or by inserting circuitry to avoid (remove) a constant input.

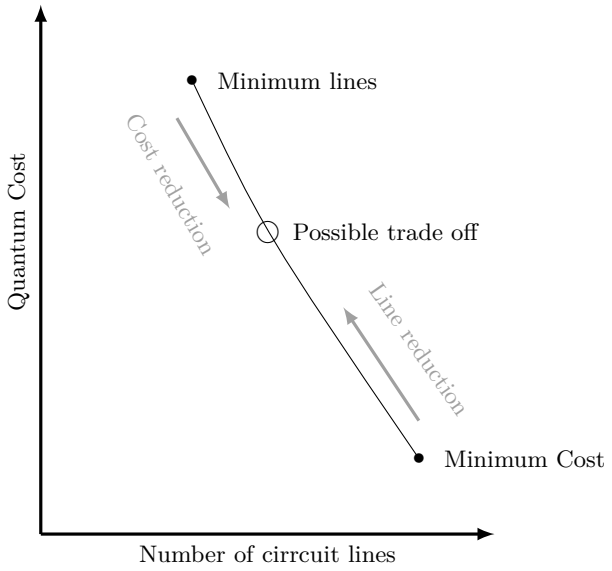


Figure 2.11: Gate costs vs. circuit lines.

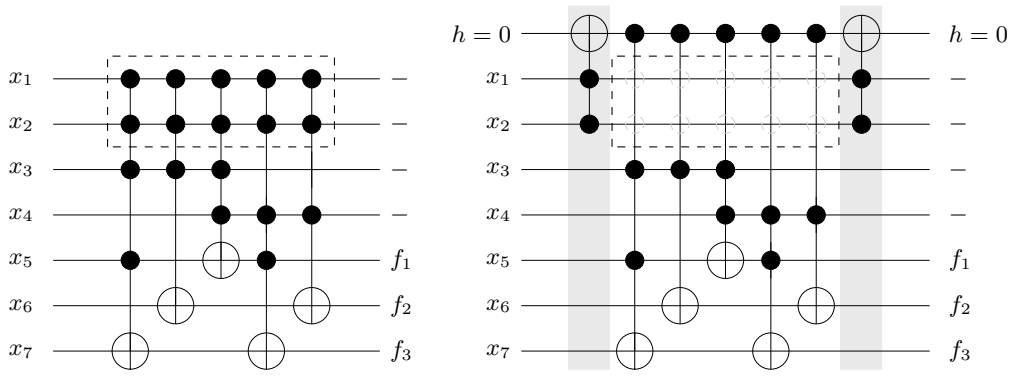
2.3.3.1 Reducing Gate Cost by Adding Lines

An observation made in [42] is exploited for reducing gate costs. Here, it was observed that many reversible circuits are composed of cascades of gates with several common control lines. As reviewed in Section 2.2.5.3, the costs of single gates depend on their respective number of control lines. Hence, buffering the results of common control conditions of a cascade of gates enables the reduction in the number of required control lines in each gate. As a result, the costs of each gate and the costs of the entire circuit are decreased significantly.

Improvement is only possible if the total costs of the two added gates are less than the costs saved by buffering the common control lines. Further, a free ancillary line must be available. This is either already the case (e.g., when a constant circuit line is required anyway for the realization of other parts in the circuit) or can explicitly be added by the designer to enable the reduction.

Following this concept, the cost of a circuit G can be moderated using a helper line as follows:

1. Determine cascades of gates $g_{1_{(C_1,T_1)}} \dots g_{k_{(C_k,T_k)}}$ which satisfy the following criteria:
   (a) The gates in the cascade have a common set C' of control lines, i.e., $C_i \supseteq C'$ for $1 \leq i \leq k$.
   (b) The value of the common control lines is not modified within this cascade, i.e., $C' \cap g_i = \emptyset$ for $1 \leq i \leq k$.
2. Create a new cascade $g_{0_{(C',\{h\})}}\ g_{1_{((C_1 \setminus C') \cup \{h\},T_1)}}\ \cdots\ g_{1_{((C_k \setminus C') \cup \{h\},T_k)}}\ g_{k+1_{(C',\{h\})}}$.
3. If a free circuit line h is available and the new cascade is cheaper than the original cascade, then replace the original cascade with the new.



(a) Original realisation (b) Revised realisation

Figure 2.12: Cost-aware synthesis.

Example 8. *Figure 2.12 demonstrates how the helper line concept is applied to reduce gate cost. Figure 2.12(a) includes a reversible circuit with five gates that has two control lines in common* $C' = \{x_1, x_2\}$*. A constant helper line* ($h = 0$) *is applied, and a new Toffoli gate is appended at the beginning of the circuit with h as the target and lines* C' *as controls, i.e.,* $g_{0_{(\{x_1,x_2\},h)}}$*. A similar gate is appended at the end of the circuit to inversely compute the ancillary helper line h, i.e.,* $g_{6_{(\{x_1,x_2\},h)}}$*. the circuit with the helper line, as shown in Figure 2.12(b), has a quantum cost metric of* ($Q_c = 59$) *(i.e., low cost) as compared to its equivalent with no helper line (Figure 2.12(a)), which has a quantum cost of* ($Q_c = 113$)*.*

Using helper lines is not limited to just one line as two helper lines may achieve a further reduction in the gate cost [42].

2.3.3.2 Reducing Lines

As mentioned in Section 2.3.2, the major drawback associated with hierarchical synthesis approaches is the high number of circuit lines. Precisely speaking, constant inputs are massively applied to the circuit to interconnect sub-circuits (sub-functions) together in realising the overall function.

In such cases, realising circuits with fewer lines is a more critical issue than the cost metrics, where the gate cost increase is justified by reducing the number of circuit lines. Consequently, hierarchical synthesis approaches with line-awareness are required to maintain the scalability while realising a circuit with fewer lines whenever possible.

The general idea of this optimisation approach exploits special circuit structures often occurring in circuits generated by hierarchical synthesis approaches [59]. Particularly, lines with constant inputs that end as garbage outputs are considered candidates for this optimisation. Since the value of a garbage output does not matter (a ”don’t care” signal), this approach might offer a possibility to merge the line with the constant input producing the garbage output. If it is possible to modify the circuit so that a garbage output returns a constant value instead of an arbitrary value, then the line can be reused in the rest of the circuit. As a result, one constant input line can be removed [60].

Definition 4. Re-computation *of a garbage line is made by applying the inverse computation to this line to obtain its initial value (constant input).*

It is possible to achieve significant reductions in the overall number of lines by re-computing and reusing lines, which might be applied on the same line more than once within the circuit. Variations of this arrangement are applied in different chapters in this work for realising operations with line-awareness.

2.4 Summary

This chapter provides a brief introduction of the necessary background on reversible circuits.

A reversible Boolean function is a bijection, i.e., a mapping where the output is uniquely determined from input and the vice versa. To satisfy this, (1) the number of input-bits should be equal to the number of output-bits, and (2) no output pattern is repeated in the truth table.

Irreversible functions can be embedded within reversible functions. This embedding requires additional bits to be added to the output pattern (garbage), such that no pattern is repeated. This garbage implies adding constant bits to the input side to make the total number of inputs equal to the total number of outputs. The embedding is optimal if only the mathematically necessary number of garbage outputs are used.

The conventional elementary gates, such as AND and, OR are not reversible. In the reversible computational paradigm, another set of gates are defined, and in this work, the reversible Toffoli and Fredkin gates are used.

Reversible circuits are cascades of reversible gates that are combined to compute certain functions. Neither feed-backs nor fan-outs are allowed directly in the reversible circuit paradigm. A circuit inverse computes the inverse function and is realised simply by reversing the order of the gates in the cascade.

The quality of a reversible circuit is measured using the number of lines as well as the total gate cost. The cost is measured either by the gate count, the transistor cost or the quantum cost. Circuit metrics (lines and cost) are shown to be trade-offs as there exist arrangements to reduce the gate cost by adding lines, and other arrangements that add gates (i.e., extra cost) to reduce some lines from the circuits.

Reversible circuit synthesis approaches are categorized into; (1) minimal line approaches, which result in fewer lines but high costs, and (2) hierarchical (decomposition) approaches, which have higher scalability but result in circuits with more lines. HDL-based synthesis is a hierarchical approach that offers high scalability in conventional as well as reversible circuit design.

Chapter 3

SyReC Specification and Synthesis of Reversible Circuits

SyReC is a dedicated reversible hardware description language that facilitates scalable synthesis through common HDL means. It allows for the specification and automatic synthesis of complex reversible circuits. SyReC was first reported in [61] to specify reversible circuits at a higher level of abstraction, in particular for the design of complex functionality, such as a RISC CPU [25]. For such designs, SyReC outperforms currently applied description means, including truth tables, permutations, and decision diagrams.

Hierarchical approaches to reversible circuit synthesis, including SyReC, typically result in circuits with a large number of lines. This sever drawback highlights the need for line-aware synthesis, which is tackled with a revised SyReC-based synthesis configured to realise the desired circuit with fewer lines [62]. Despite being less critical for this approach, the cost-aware configuration has also been proposed to reduce the gate cost associated with certain cases [24].

In this chapter, we introduce the SyReC as the state-of-the-art and cover in detail general concepts, specification syntax and semantics, its special hardware related properties and constraints, and the synthesis of SyReC specifications. [1]

3.1 SyReC: The Language

Since all (valid) SyReC programs are inherently reversible, the reversibility of the specification is simultaneously ensured. The general concepts to achieve this are summarised in the first part of this section followed by an explanation of the syntax and semantics of all SyReC description means.

[1] Comprehensive reviews with additional details on SyReC are found in [24, 63].

3.1.1 General Concepts

To ensure reversibility in its description, SyReC adapts established concepts from the previously introduced reversible programming language, *Janus* [64], and is additionally modified by hardware-related language constructs since it targets the description of reversible circuits. The

1. **Reversible Assignments:** Being one of the most elementary language constructs, variable assignments, such as those used in most imperative languages, are irreversible and cannot be part of a reversible language. The concept of *reversible assignments* (also called reversible updates) is used as an alternative. Reversible assignments have the form $v \oplus = e$ with $\oplus \in \{\hat{}, +, -\}$ such that the variable v does not appear on the right-hand side expression e. Although SyReC is limited to the set of operators in $\oplus$, any operator f can be used for the reversible assignment if there exists an inverse operator f^{-1} such that
$$v = f^{-1}(f(v, e), e) \tag{3.1}$$
for all variables v and all expressions e. Note that '+' (addition) is inverse to '−' (subtraction) and vice versa, and '^' (bit-wise exclusive OR) is inverse to itself. When executing the program in reverse order, all reversible assignment operators are replaced by their inverse operators.

2. **Expressiveness:** Due to the construction of the reversible assignment, the right-hand side expression can also be irreversible and compute any operation. The most common operations are directly applicable using a wide variety of syntax, including arithmetic (+, *, /, %, *>), bit-wise (&, |, ^), logical (&&, ||), and relational (<, >, =, !=, <=, >=) operations. The reversibility is ensured, since the input values to the operation are also given to the inverse operation when reverting the assignment (see Equation (3.1)). For example, to specify a multiplication a*b, a new free signal c must be introduced to store the result, i.e., c^=(a*b) is applied.

3. **Reversible Control Flow** A reversible data flow is ensured due to the assignment operation mentioned above, and the control flow is made bijectively executable in a similar fashion. This becomes particularly manifest in conditional statements. In contrast to non-reversible languages, SyReC requires an additional *fi*-condition for each *if*-condition which is applied as an assertion. This *fi*-condition is required, since a conditional statement may not be computed in both directions using the same condition, i.e., it cannot be ensured that the same block (*then*-block or *else*-block) is processed when computing an *if*-statement in the reverse direction. As a solution, a corrected *fi*-condition asserted when computing the statement in the reverse direction is added to ensure a consistent execution semantic. This language principle is illustrated in detail in the next section.

4. **Hardware Description Properties:** Since SyReC is used for the synthesis of reversible circuits, it obeys some HDL related properties:

   - The single data-type is a circuit signal with parameterised bit-width.
   - Access to single bits ($x.N$), a range of bits ($x.N{:}N$), and the size ($\#x$) of a signal is provided.
   - Since loops must be completely unrolled during synthesis, the number of iterations has to be available before compilation. So, dynamic loops (defined by expressions) are not allowed.
   - Additional operations used in hardware design (e.g., shifts '<<' and '>>') are provided.

   The implementation of these general concepts for the SyReC syntax is illustrated in detail using the EBNF in Figure 3.1.

Table 3.1: Signal access modifiers and implied circuit properties.

Modifier	Constant Input	Garbage Output	State	Initial Value
in	–	yes	no	given by primary input
out	0	no	no	0
inout	–	no	no	given by primary input
wire	0	yes	no	0
state	–	no	yes	given by pseudo-primary input

3.1.2 Module and Signal Declarations

Each SyReC specification (denoted by $\langle program \rangle$ in line 1 in Figure 3.1) consists of one or more *modules* (denoted by $\langle module \rangle$ in line 2). A module is introduced with the keyword *module* and includes an identifier (represented by a string as defined in line 23), a list of parameters representing global signals (denoted by $\langle parameter - list \rangle$ in line 3), local signal declarations (denoted by $\langle signal - list \rangle$ in line 5), and a sequence of statements (denoted by $\langle statement - list \rangle$ in line 7). The top-module of a program is defined by the special identifier *main*. If no module with this name exists, the last module declared is used as the top-module instead.

SyReC uses a *signal* representing a non-negative integer as its sole data type. Round brackets can optionally define the bit-width of signals after the signal name (line 6). If no bit-width is specified, a default value of 32-bit is assumed. For each signal, an *access modifier* must be defined. For a parameter signal (used in a module declaration), this can be either *in*, *out*, or *inout* (line 4).

Program and Modules

1 ⟨*program*⟩ ::= ⟨*module*⟩ {⟨*module*⟩}
2 ⟨*module*⟩ ::= **'module'** ⟨*identifier*⟩ **'('** [⟨*parameter-list*⟩] **')'** {⟨*signal-list*⟩} ⟨*statement-list*⟩
3 ⟨*parameter-list*⟩ ::= ⟨*parameter*⟩ {**','** ⟨*parameter*⟩}
4 ⟨*parameter*⟩ ::= (**'in'** | **'out'** | **'inout'**) ⟨*signal-declaration*⟩
5 ⟨*signal-list*⟩ ::= (**'wire'** | **'state'**) ⟨*signal-declaration*⟩ {**','** ⟨*signal-declaration*⟩}
6 ⟨*signal-declaration*⟩ ::= ⟨*identifier*⟩ {**'['**⟨*int*⟩**']'**} [**'('**⟨*int*⟩**')'**]

Statements

7 ⟨*statement-list*⟩ ::= ⟨*statement*⟩ {**';'** ⟨*statement*⟩}
8 ⟨*statement*⟩ ::= ⟨*call-statement*⟩ | ⟨*for-statement*⟩ | ⟨*if-statement*⟩ | ⟨*unary-statement*⟩ | ⟨*assign-statement*⟩ | ⟨*swap-statement*⟩ | ⟨*skip-statement*⟩
9 ⟨*call-statement*⟩ ::= (**'call'** | **'uncall'**) ⟨*identifier*⟩ **'('** (⟨*identifier*⟩ {**','** ⟨*identifier*⟩}) **')'**
10 ⟨*for-statement*⟩ ::= **'for'** [[**'$'** ⟨*identifier*⟩ **'='**] ⟨*number*⟩ **'to'**] ⟨*number*⟩ [**'step'** [**'-'**] ⟨*number*⟩] ⟨*statement-list*⟩ **'rof'**
11 ⟨*if-statement*⟩ ::= **'if'** ⟨*expression*⟩ **'then'** ⟨*statement-list*⟩ **'else'** ⟨*statement-list*⟩ **'fi'** ⟨*expression*⟩
12 ⟨*assign-statement*⟩ ::= ⟨*signal*⟩ (**'^'** | **'+'** | **'-'**) **'='** ⟨*expression*⟩
13 ⟨*unary-statement*⟩ ::= (**'~'** | **'++'** | **'--'**) **'='** ⟨*signal*⟩
14 ⟨*swap-statement*⟩ ::= ⟨*signal*⟩ **'<=>'** ⟨*signal*⟩
15 ⟨*skip-statement*⟩ ::= **'skip'**
16 ⟨*signal*⟩ ::= ⟨*identifier*⟩ {**'['** ⟨*expression*⟩ **']'**} [**'.'** ⟨*number*⟩ [**':'** ⟨*number*⟩]]

Expressions

17 ⟨*expression*⟩ ::= ⟨*number*⟩ | ⟨*signal*⟩ | ⟨*binary-expression*⟩ | ⟨*unary-expression*⟩ | ⟨*shift-expression*⟩
18 ⟨*binary-expression*⟩ ::= **'('** ⟨*expression*⟩ (**'+'** | **'-'** | **'^'** | **'*'** | **'/'** | **'%'** | **'*>'** | **'&&'** | **'||'** | **'&'** | **'|'** | **'<'** | **'>'** | **'='** | **'!='** | **'<='** | **'>='**) ⟨*expression*⟩ **')'**
19 ⟨*unary-expression*⟩ ::= (**'!'** | **'~'**) ⟨*expression*⟩
20 ⟨*shift-expression*⟩ ::= **'('** ⟨*expression*⟩ (**'<<'** | **'>>'**) ⟨*number*⟩ **')'**

Identifier and Constants

21 ⟨*letter*⟩ ::= (**'A'** | ... | **'Z'** | **'a'** | ... | **'z'**)
22 ⟨*digit*⟩ ::= (**'0'** | ... | **'9'**)
23 ⟨*identifier*⟩ ::= (**'_'** | ⟨*letter*⟩) {(**'_'** | ⟨*letter*⟩ | ⟨*digit*⟩)}
24 ⟨*int*⟩ ::= ⟨*digit*⟩ {⟨*digit*⟩}
25 ⟨*number*⟩ ::= ⟨*int*⟩ | **'#'** ⟨*identifier*⟩ | **'$'** ⟨*identifier*⟩ | (**'('** ⟨*number*⟩ (**'+'** | **'-'** | **'*'** | **'/'**) ⟨*number*⟩ **')'**)

Figure 3.1: Syntax of the hardware description language SyReC.

Local signals can either work as internal signals (denoted by *wire*) or, in the case of sequential circuits as state signals[2] (denoted by *state*; line 5). The access modifier affects properties in the synthesised circuits as summarized in Table 3.1. Signals can be grouped into multi-dimensional arrays of constant length using square brackets after the signal name and before the optional bit-width declaration (line 6).

Example 9. *Figure 3.2 shows an exemplary module* `(myCircuit)` *declaration possible in SyReC, including one* ***in*** *signal* `(a)` *with a single bit, a four-element array of* ***inout*** *signals* `(x)` *each with 16 bit-width and an* ***out*** *signal* `(y)` *with a default bit-width (32 bit). Also, an internal signal (****wire****)* `(auxSignal)` *is declared with 16 bits and as well as a state signal* `(stateSignal)` *with a default bit-width of 32 bit.*

```
module myCircuit(in opr (1), inout x [4] (16), out y)
        wire auxSignal(16)
        state stateSignal
```

Figure 3.2: Exemplary module and internal signal and state declarations.

3.1.3 Statements

Statements include call and uncall of other modules, loops, conditional statements, and various data operations (i.e., reversible assignment operations, unary operations, and swap statements as in line 8). The empty statement can explicitly be modelled using the *skip* keyword (line 15). Statements are separated by semicolons (line 7). Signals within statements are denoted by $\langle signal \rangle$ allowing access to the entire signal (e.g., `x`), a certain bit (e.g., `x.4`), or a range of bits (e.g., `x.2:4` as in line 16). The bit-width of a signal can also be accessed (e.g., `#x` as in line 25).

3.1.3.1 Signal Update Statements

Reversible signal update statements include the reversible signal assignment(denoted by $\langle assign - statement \rangle$), unary statements (denoted by $\langle unary - statement \rangle$), and the swap statement (denoted by $\langle swap - statement \rangle$) as defined in line 12 to 14. The semantics of these statements are summarised in Table 3.2, whereby x, y denote signals and e denotes expressions. Since these statements perform only reversible operations, they may assign new values to signals. Therefore, the respective signal(s) to be modified must not appear in the expression on the right-hand side.

[2]Note that depending on the application feedback, the corresponding state signals might not be allowed in reversible circuits. Nevertheless, SyReC supports this concept in principle. For a detailed discussion on reversible sequential circuits, refer to [65, 66].

Table 3.2: Assignment, unary, and swap statements in SyReC.

Assignment	Semantic		Inverse
$x \hat{}= e$	**Bit-wise XOR** assignment of e to x,	$x := x \hat{} e$	$x \hat{}= e$
$x \mathrel{+}= e$	**Increase** by value of e to x,	$x := x + e$	$x \mathrel{-}= e$
$x \mathrel{-}= e$	**Decrease** by value of e to x,	$x := x - e$	$x \mathrel{+}= e$
$\tilde{}= x$	**Bit-wise inversion** of x,	$x := x$	$\tilde{}= x$
$++= x$	**Increment** of x,	$x := x + 1$	$--= x$
$--= x$	**Decrement** of x,	$x := x - 1$	$++= x$
$x <=> y$	**Swapping** value of x with value of y,	$x := y$ *and* $y := x$	$x <=> y$

3.1.3.2 Conditional Statements

Conditional statements (defined in line 11) need an expression to be evaluated followed by the respective *then*- and *else*-block. Each of these blocks is a sequence of statements. In a forward computation, the *then*-block is executed if and only if the *if*-expression evaluates to 1; otherwise, the *else*-block is executed. To ensure reversibility, a conditional statement is terminated by a *fi* together with an adjusted expression. In a backward (inverse) computation, the *fi*-expressions decide whether the *then* or the *else*-block is reversibly executed. If neither the *then* nor the *else*-block modifies an input value of the conditional expression, then the *if* and *fi* expressions are identical.

```
if (b = 5) then
  x += y
    // executed if b = 5
else
  x -= y
    // executed if b != 5
fi (b = 5);
```

(a) Identical if- and fi-conditions

```
if (b = 5) then
  b += y
    // executed if b = 5 (fwd)
    // or b = 5 + y (bwd)
else
  x -= y
    // executed otherwise
fi (b = (5 + y))
```

(b) Different if- and fi-conditions

Figure 3.3: Conditional statements in SyReC.

Example 10. *Figure 3.3(a) shows a conditional statement in SyReC, which does not modify any of the inputs of the forward condition (signal b in this case). Hence, the* if *and* fi *expressions are identical. In contrast, the* then*-block of the conditional statement (Figure 3.3(b)) modifies the value of signal b. So, a suitable* fi*-expression different from the* if*-expression must be provided to ensure correct execution semantics in both directions.*

3.1.3.3 Loops

An iterative execution of a block is defined by loops (defined in line 10). The number of iterations must be available prior to the compilation, i.e., dynamic loops are not allowed. Therefore, to fix integer values, for example, the bit-width of a signal, or internal (local) *$*-variables can be applied. Furthermore, the current value of the internal counter variables can be accessed during the iterations. Using the optional keyword *step*, the iteration itself can be also modified. A loop is terminated by *rof*.

```
for $counter = 1 to 10 step 2 do
        // statements
        // the loop iterates 5 times
        // (i.e., $counter is set to 1, 3, 5, 7, and 9 only)
rof
```

Figure 3.4: Exemplary loops in SyReC.

Example 11. *Figure 3.4 shows an exemplary loop description possible in SyReC. The loop with a counter variable* `$counter` *takes the values* `1` *to* `10` *with an increment step of* `2`*.*

3.1.3.4 Call and Uncall of Modules

Hierarchic descriptions are realised in SyReC by using modules which can be called and uncalled. For this purpose, the keyword *call* (*uncall*) must be applied together with the identifier of the module to be called along with its parameters (line 9). Call executes the selected module in the forward direction, while uncall executes the selected module backwards.

```
module max(inout a, inout b, out m)
        if (a < b) then
                m ^= a
        else
                m ^= b
        fi (a < b)

module main(inout x, inout y, inout z, out max)
        wire temp
        call max(x, y, temp)
        call max(temp, z, max)
```

Figure 3.5: Calling a module.

Example 12. *Figure 3.5 presents a SyReC description of two modules:* `main`*, which is the top-level module and* `max`*, which is a sub-module called (instantiated) twice within* `main`*.*

3.1.4 Expressions

Expressions, as defined in lines 17 to 20, are applied on the right-hand side of assignment statements or as branching conditions in *if* and *fi* statements. Since expressions do not modify the values of any signal, non-reversible operations can also be applied in expressions without jeopardising the reversibility. Because of this, a wide range of different description means is provided. Table 3.3 lists the semantics for all operations which can be used in expressions, whereby e, f denote sub-expressions and N denotes natural numbers.

Table 3.3: Expressions in SyReC.

Operation	Semantic
e + f	Addition of e and f
e − f	Subtraction of e and f
e * f	Lower bits of multiplication of e and f
e *> f	Upper bits of multiplication of e and f
e / f	Division of e and f
e % f	Remainder of division of e and f
e ^ f	Bit-wise XOR of e and f
e & f	Bit-wise AND of e and f
e | f	Bit-wise OR of e and f
~e	Bit-wise inversion of e
e && f	Logical AND of e and f
e || f	Logical OR of e and f
!e	Logical NOT of e
e < f	True, if and only if e is less than f
e > f	True, if and only if e is greater than f
e = f	True, if and only if e equals f
e != f	True, if and only if e not equals f
e <= f	True, if and only if e is less or equal to f
e >= f	True, if and only if e is greater or equal to f
e << N	Logical left shift of e by N
e >> N	Logical right shift of e by N

```
c ^= (a * b);  // c := a*b if c is a new free signal
x.0 ^= ((a > 3) && (b != 0));
x.1:3 ^= (c.0:2 | 4);
if (a = b) then c+= (a % 2) else c -= (b / 2) fi (a = b)
```

Figure 3.6: Application of expressions.

Example 13. *Figure 3.6 shows statements for expressions that demonstrate the range of description means available in SyReC. Although the language is restricted to ensure reversibility (e.g., statements such as* `c = a * b` *are not allowed), common functionality can easily be specified (e.g., with a new free signal c, such as* `c ^= (a * b)` *). Despite the usage of non-reversible operations in Figure 3.6, all statements still can be executed in both directions.*

3.2 Synthesis of SyReC Specifications

In SyReC, it is possible to specify reversible circuits on a higher level of abstraction. For the design of complex functionality, SyReC clearly outperforms currently applied description means, such as truth tables, permutations, and decision diagrams. The next step is to realise SyReC specifications as reversible circuits.

To synthesise a given SyReC specification, a hierarchical synthesis method is developed. Existing realisations, so-called *building blocks*, of the individual statements and expressions, are used and combined so that the desired circuit is realised. More precisely, SyReC synthesis (1) traverses the entire program and (2) adds cascades of reversible gates to the circuit realising each statement or expression.

Modules are synthesised independently of each other and afterwards cascaded according to the respective *call* and *uncall* statements. All signals are realised by buses of common reversible circuit lines with the specified bit-width. In the following, the individual mappings of the statements to the respective reversible cascades are described. We distinguish between the synthesis of assignment statements (including unary and swap statements), expressions, and control logic including call and uncall, loops, and conditional statements.

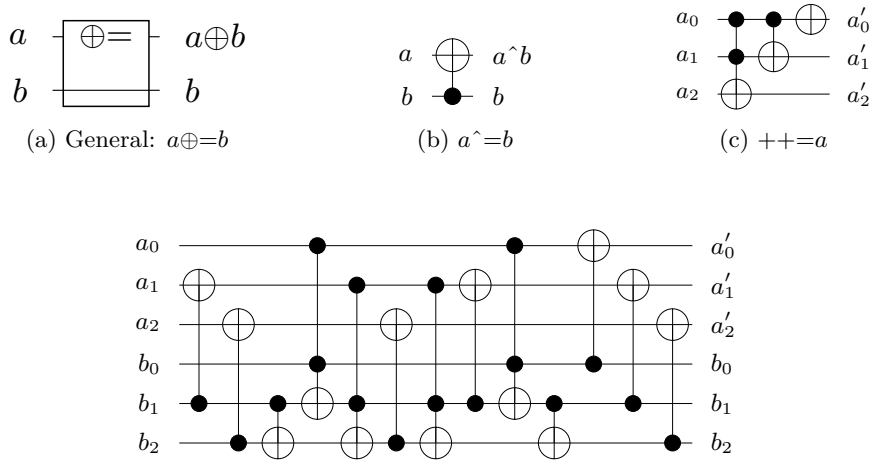
3.2.1 Synthesis of Assignment Statements

As introduced in Section 3.1.3.1, assignment statements in SyReC must be reversible. As a consequence, signal values are not overwritten but rather updated with a new value such that the old value can still be recovered by applying the inverted assignment operation. The notation, as depicted in Figure 3.7(a), denotes such operations in circuit structures. Solid lines that cross the box represent the signals(s) on the right-hand side of the statement, i.e., the signal(s) whose values are preserved.

The most straightforward reversible assignment operation is the bit-wise XOR (e.g., $a \hat{} = b$). For 1-bit signals, this operation can be synthesised by a single Toffoli gate, as shown in Figure 3.7(b). If signals with a bit-width greater than 1 are applied, for each bit a Toffoli gate is applied analogously.

To synthesise the increase operation ($a \mathrel{+}= b$), a modified addition network is added. In the past, several reversible circuit realisations for addition were investigated, and it is well known that the minimal realisation of a 1-bit adder consists of four Toffoli gates [67]. Thus, cascading the required number of 1-bit adders is a possible realisation. However, since every 1-bit adder also requires one constant input, this is a poor solution with respect to circuit lines. In contrast, heuristic realisations exist that require a fewer number of additional lines [68]. Since the increase operation, unlike addition, is reversible, it can even be synthesised without additional lines. Such a realisation is used in our approach. A corresponding cascade for a 3-bit increase is depicted in Figure 3.7(d).

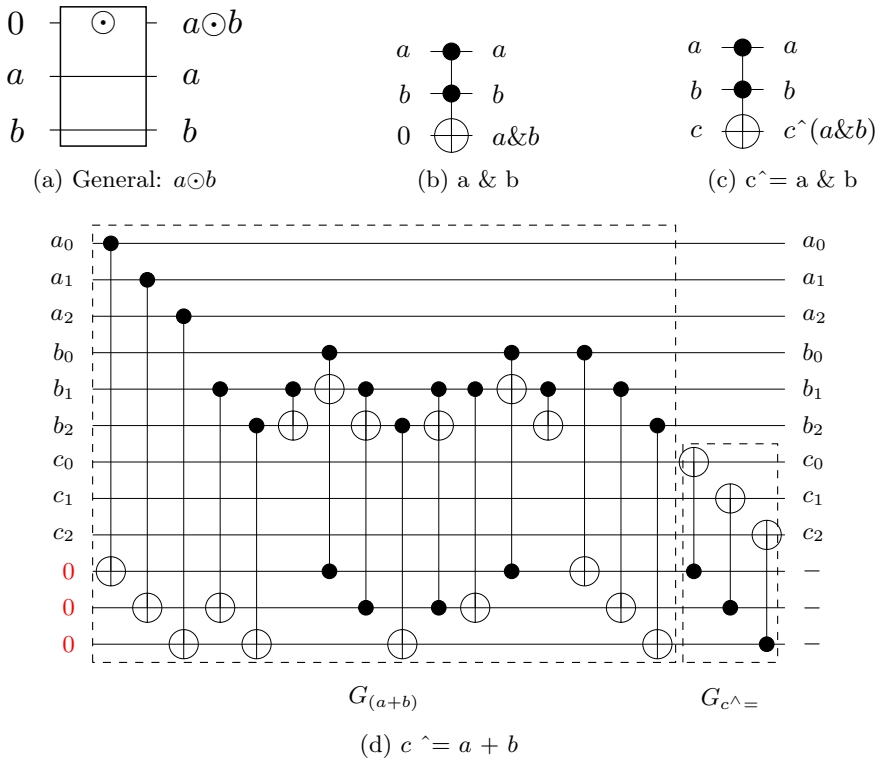
The mapping for the decrease operation ($a \mathrel{-}= b$) is the inverse of the increase operation. So, the same realisation depicted in Figure 3.7(d) is used in reverse.



(a) General: $a \oplus = b$ (b) $a \hat{} = b$ (c) $++ = a$

(d) $a += b$ (for 3-bit variables)

Figure 3.7: Synthesis of assignment statements.



(a) General: $a \odot b$ (b) a & b (c) c^= a & b

(d) $c \hat{} = a + b$

Figure 3.8: Synthesis of expressions.

Finally, the realisations for the unary and swap statements are straightforward. The bit-wise inversion ($\tilde{} = a$) is realised by adding a NOT gate to each circuit line representing a bit of a. Similarly, a swap ($a <=> b$) is realised by adding a SWAP gate to the corresponding circuit lines of a and b. To synthesise an increment ($++ = a$), a cascade, as depicted in Figure 3.7(c), is applied. A decrement ($-- = a$) is realised by using the same cascade in reverse.

3.2.2 Synthesis of Expressions

Expressions include operations that are not necessarily reversible so that their inputs must be preserved to allow a (reversible) computation in both directions. To denote such operations, the notation depicted in Figure 3.8(a) is used. Again, solid lines represent the signals(s) whose values are preserved (i.e., the input signals).

Synthesis of irreversible functions in reversible logic is not new, so for most of the respective operations, reversible circuit realizations already exist. Additional lines with constant inputs are applied to make an irreversible function reversible [35,36]. As an example, Figure 3.8(b) shows a reversible gate that realises an AND operation. As can be seen, this requires one additional circuit line with a constant input $'0'$. Similar mappings exist for all other operations.

Since expressions can be applied together with assignment statements (e.g., $c\hat{}=a\&b$), sometimes a more compact realisation is possible. More precisely, additional constant circuit lines can be saved (for some statements), if the result of an expression is applied to an assignment statement. As an example, Figure 3.8(c) shows the realisation for $c\hat{}=a\&b$ where no constant input is needed, but the circuit line representing c is used instead. However, such a simple "combination" is not possible for all statements. As an example, Figure 3.8(d) shows 3-bit addition whose result is applied to a bit-wise XOR, i.e., $c\hat{}=a+b$. Here, removing the constant lines and directly applying the XOR operation on the lines representing c would lead to an incorrect result because intermediate results are stored at the lines representing the sum. Since these values are reused later, performing the XOR operation "in parallel" would destroy the result. Thus, to have a combined realisation of a bit-wise XOR and an addition, a precise embedding for this case must be generated. Since determining the respective embedding and circuits for arbitrary combinations of statements and expressions is a cumbersome task, constant lines are applied to realise the respective functionality. However, in Section 3.3, an extended synthesis scheme is presented that removes many of these additional lines.

Using the building blocks for reversible signal statements and expressions as introduced above, it is possible to synthesise reversible circuits specified in SyReC automatically. More precisely, the following two steps are performed for each statement:

1. Compose a sub-circuit $G_{\odot}$ realising all the expressions in a statement using the respective building blocks. The result of an expression is buffered using an additional line.

2. Compose a sub-circuit $G_{\oplus}$ realising the overall statement using the existing building blocks of the statement together with the buffered results of the expressions.

Hence, the resulting circuits have a structure as shown in Figure 3.9, i.e., cascades of building blocks for the respective assignment statements and their expressions results.

Obviously, this leads to a significant number of additional circuit lines with constant inputs used to buffer intermediate results of the expressions. The precise number of additional circuit lines increases with respect to the complexity of the expression. Usually, a large number of circuit lines is seen as a disadvantage.

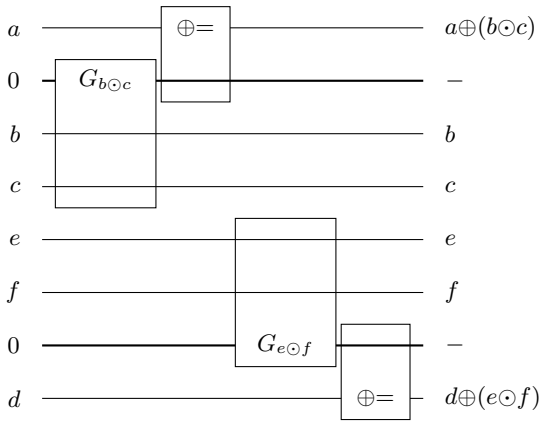
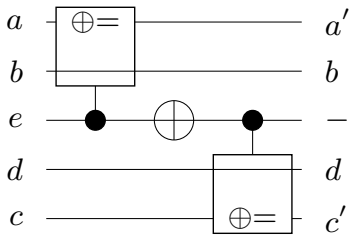


Figure 3.9: Resulting circuit structure.

```
if e
 then
   a ⊕= b
 else
   c ⊕= d
fi e
```

(a) Code



(b) Circuit realization

Figure 3.10: Synthesis of conditional statements.

3.2.3 Synthesis of the Control Logic

Finally, the synthesis of control logic is considered. Module calls, uncalls, and loops are realised straightforwardly. Loops are realized by simply cascading (i.e., unrolling) the respective statements within a loop block for each iteration. Since the number of iterations must be fixed (see Section 3.1.3.3), this results in a finite number of statements subsequently processed. Call and uncall of modules are handled similarly where the respective statements in the modules are cascaded.

To realise conditional statements, such as *if*-statements as introduced in Section 3.1.3.2, the statements in the *then*- and *else*-block are mapped to reversible cascades with an additional control line added to all gates (see Figure 3.10(b)). Thus, the respective operations of the statements in the *then* or (*else*-block are computed if and only if the result of the expression (stored in signal e) is 1 or 0, respectively. A NOT gate is applied to flip the value of e so that the gates of the *else*-block can also be "controlled".

3.3 Circuit Optimization

Hierarchical synthesis approaches do not guarantee optimal circuits, which keeps a space opened for refinement. Both circuit trade-off metrics may be considered as optimisation objectives.

3.3.1 Line-aware Synthesis of SyReC Specifications

To realise SyReC specifications with a smaller number of additional circuit lines, an extended synthesis scheme is presented in this section (based on [62]). The idea is to use the same building blocks as introduced in the previous section but to undo intermediate results of the expressions as soon as they are no longer needed. A similar idea (for reversible software programs) was previously proposed in [69], which enables that reuse of circuit lines which previously occupied by expressions.

In the following, the concept of this scheme is illustrated before the extended synthesis is described in detail for all possible SyReC statements. Next, the necessary amount of additional circuit lines is discussed.

3.3.1.1 General Concept

The extended synthesis approach follows the scheme as introduced in Section 3.2.2, but is extended by an additional third step:

3. Add the inverse circuit from Step 1, $G_{\odot}^{-1}$, to the circuit to reset the circuit lines buffering the result of the expressions to the constant $'0'$.

Example 14. *Consider the two following generic HDL statements:*

```
a ⊕= (b ⊙ c);
d ⊕= (e ⊙ f);
```

Figure 3.11 sketches the resulting circuit after applying the extended synthesis scheme. The first two sub-circuits $G_{b\odot c}$ and $G_{a\oplus=b\odot c}$, ensure that the first statement is realised. This is equivalent to the scheme proposed in Section 3.2 and leads to additional lines with constant inputs (highlighted thick). Afterwards, a further sub-circuit, $G_{b\odot c}^{-1}$, is applied. Since $G_{b\odot c}^{-1}$ is the inverse of $G_{b\odot c}$, this sets the circuit lines buffering the result of $b \odot c$ back to the constant $'0'$. As a result, these circuit lines can be reused to realise the following statements as illustrated for $d\oplus=e \odot f$ in Figure 3.11.

3.3.1.2 Resulting Synthesis Scheme

Following the proposed concept, each statement can be realised with zero garbage outputs. In the following, the precise realisation of this scheme is detailed for each possibly affected statement. The unary statements, the swap-statement (<=>), and the skip statement are not considered here as they are realised without additional circuit lines.

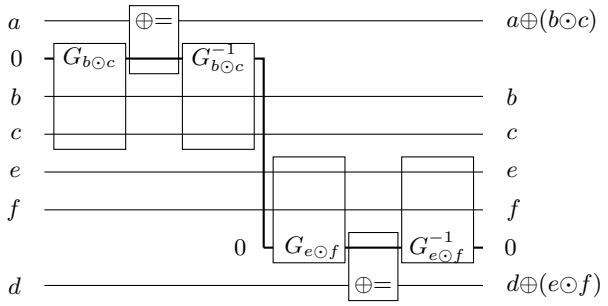


Figure 3.11: Line reduction.

Assignment Statements To realise statements of the form $a \oplus\!= e$ with e being an arbitrary expression, the respective building blocks are orchestrated as illustrated in Figure 3.11. First, a sub-circuit realising the expression e, i.e., the right-hand side of the statement, is created. This requires additional lines to store the result of e. Next, a sub-circuit realising the assignment operation is created as well as a sub-circuit reversing the result of e into a constant value. The latter is done by reversing the order of gates of the first sub-circuit. Finally, all three sub-circuits are composed leading to the desired realisation of the statement.

Example 15. *Figure 3.12 shows the circuit obtained by synthesising* $c \hat{}= (a + b)$ *using the extended synthesis scheme. Dashed rectangles highlight the respective sub-circuits* G_{a+b}*,* $G_{c \hat{}= a+b}$*, and* G^{-1}_{a+b}*. Since all gates considered in this work are self-inverse,* G^{-1}_{a+b} *is obtained by reversing the order of the gates of* G_{a+b}*. The target signal is updated exactly as in Figure 3.8(d), but here the three garbage lines are re-computed to the constant* $'0'$*.*

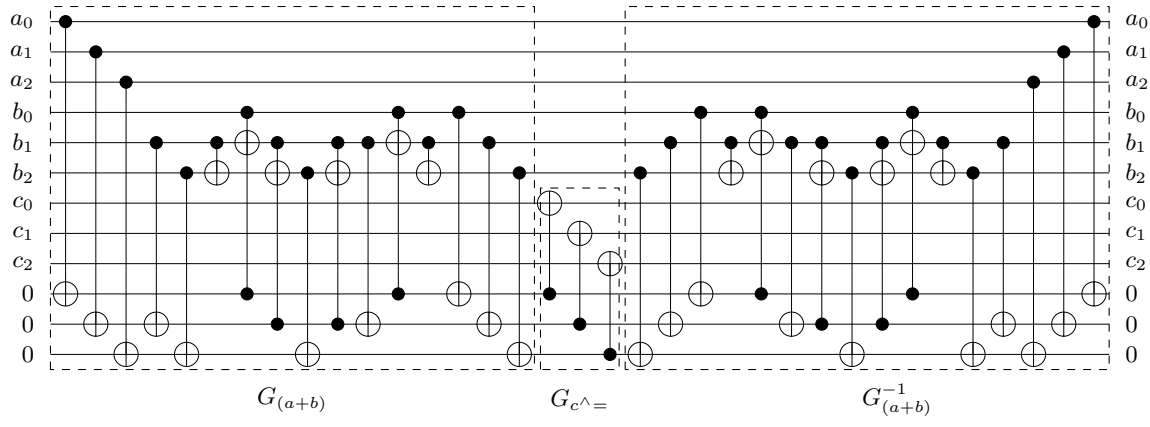


Figure 3.12: Synthesizing `c ^= (a+b)` with no garbage.

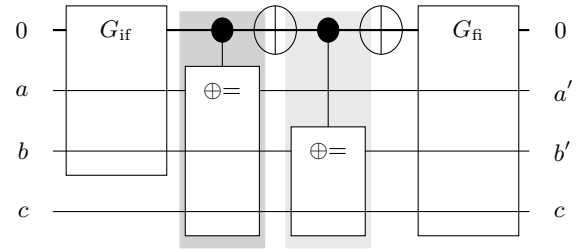
Applying this procedure, any arbitrary combination of assignment statements and expressions can be realised in a line-aware manner. That is, required additional circuit lines are ancilla lines and can be reused for other statements and operations.

Conditional Statements

As described in Section 3.2.3, conditional statements are realised by combining two sub-circuits, G_{then} and G_{else} (see Figure 3.10). Figure 3.13(b) illustrates the adjusted procedure for the synthesis of a conditional statement in the line-aware configuration of circuits. The gates needed to realise the *then*-block (*else*-block) are highlighted in dark grey (light grey). Also, a sub-circuit G_{if} evaluating the respective *if*-expression is created. The intermediate results of that expression are handled analogously to assignment statements as described above. An additional circuit line is applied to store the Boolean result of the *if*-expression and control the execution of the *then*- and *else*-block as described in Section 3.2.3. The flip on the additional line, which controls the gates of the *else*-block, is then restored by another NOT gate. Afterwards, the original constant value of that line is restored by applying a sub-circuit G_{fi} which evaluates the *fi*-expression of the statement. As defined in Section 3.1.3.2, SyReC requires the definition of a *fi*-expression that evaluates to the same Boolean value as the *if*-expression did in G_{if}.

```
if (a = b)
 then
   a ⊕= c
 else
   b ⊕= c
fi (a = (b ⊕ c))
```

(a) Code



(b) Without additional lines

Figure 3.13: Realisation of a conditional statement in line-aware SyReC synthesis.

Loops and Calls

The realisation of loops and module calls is treated in a straight forward manner exploiting the procedures proposed above. Calls are substituted by the corresponding statements inside the body of the call. Loops are realised by explicitly cascading (i.e., unrolling) the respective statements within a loop block according to the fixed and finite number of iterations.

3.3.1.3 Discussion

Applying the extended synthesis scheme, every statement is synthesised with zero garbage outputs and only additional ancilla lines. Consequently, the total number of additional lines required to realize a SyReC specification with the proposed solution can be determined by the statement that requires the largest number of additional lines to buffer intermediate results.

Example 16. *Consider a sequence of three assignment statements to be synthesised. Additionally, assume that 1, 3, and 2 circuit lines are needed to buffer the intermediate results of the respective expressions. Then, in total* $\max\{1,3,2\} = 3$ *additional circuit lines are needed to realise the statements. Figure 3.14 illustrates how these circuit lines are applied. For comparison, the synthesis scheme from Section 3.2 needs* $1+3+2 = 6$ *additional circuit lines.*

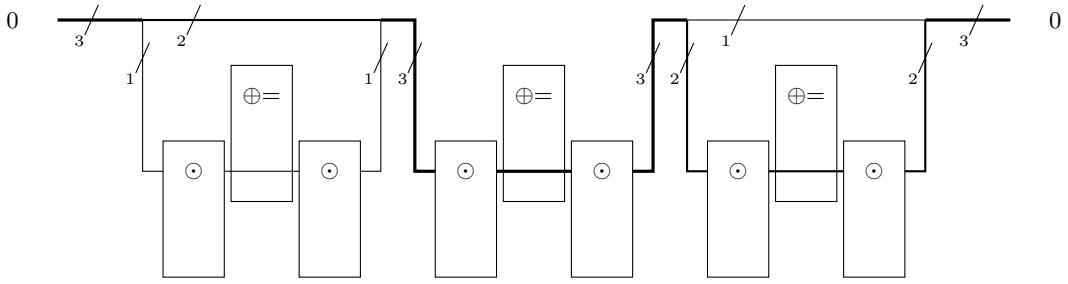


Figure 3.14: Effect of expression size.

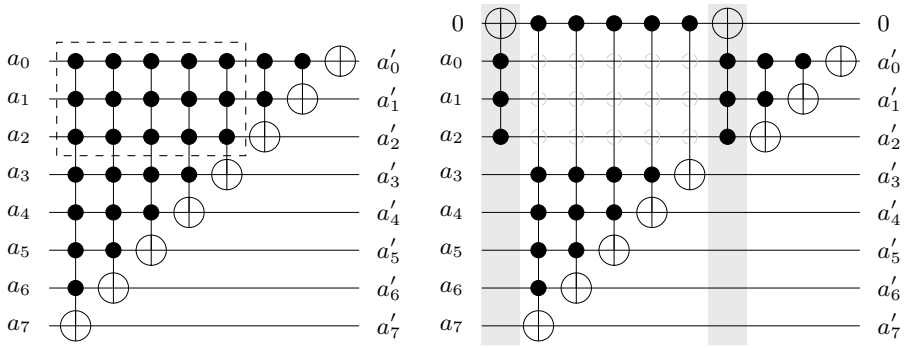
Overall, a price for the smaller number of circuit lines is an expected increase in the number of gates and, thus, in the gate costs. However, the increase in the gate costs is bounded. For example, in comparison to the synthesis scheme from Section 3.2 where the building blocks $G_{\odot}$ and $G_{\oplus}$ are applied for each assignment statement, the extended scheme uses just one more building block $G_{\odot}^{-1}$. Since $G_{\odot}^{-1}$ is the inverse of $G_{\odot}$, the circuit can, at most, double its gate cost. Overall, the resulting circuits still include additional circuit lines with constant inputs. Considering that, until today, the synthesis of complex functions as a reversible circuit with the minimal number of lines is a cumbersome task [35], the proposed solution enables keeping this number relatively small.

3.3.2 Cost-aware Synthesis of SyReC Specifications

SyReC synthesis can be refined to reduce the costs of the resulting circuits. Here, it is observed that some SyReC operations are realised as cascades of gates with several common control lines, and it is cost-worthy to apply cost reduction optimisation, as described in Section 2.3.3.1. While this arrangement was originally proposed for minimal line synthesis approaches with extremely high costs, it can still improve the hierarchical SyReC synthesis.

Example 17. *Consider an 8-bit realisation of the increment statement (++=a) as shown in Figure 3.15(a). The gates in this cascade have several common control lines, e.g.,* $C' = \{a_0, a_1, a_2\}$. *By adding two Toffoli gates* $g_{(C',\{h\})}$, *the result of the common control conditions* C' *can be buffered in an ancilla line* h *as shown in Figure 3.15(b) (the new gates are emphasized with a grey box and the line* h *is on the top). This enables all gates with control lines* $C \supseteq C'$ *to be simplified, i.e., instead of* C, *a smaller set of control lines* $(C \setminus C') \cup \{h\}$ *is sufficient (in Figure 3.15(b), the saved control lines are indicated with dashed circles). As*

a result, the costs of the gates and, hence, the costs of the overall circuit are significantly reduced. In fact, quantum costs can be improved from 431 to 116 (73%), and transistor costs can be improved from 224 to 192 (14%).



(a) Original realisation (b) Revised realisation

Figure 3.15: 8-bit realisation of the increment statement (++=a).

Similar observations can be made for other building blocks, including nested conditional statements that frequently lead to large cascades of gates with common control lines. This is because the circuit lines representing the conditional expressions control entire cascades realising the respective *then*- and *else*-blocks.

This procedure applies to both synthesis approaches, i.e., to the scheme proposed in Section 3.2 as well as to the extended scheme proposed in Section 2.3.3.1. Determining the best possible cascades for replacement is a complex task as the order in which common control lines are exploited typically has an effect. Hence, we apply this procedure only for single statements leading to local optima. As confirmed by the experiments, significant improvements are reported [24].

3.4 Summary

This chapter introduces the SyReC language, SyReC synthesis, and circuit optimisation in SyReC.

- **The SyReC language** as introduced in Section 3.1 is a language designed to describe reversible hardware with a modular style. A SyReC program is made of modules what call each other in a hierarchical structure. A module is declared similarly to a declaration of a function. The input and output signals of a SyReC module appear in the declaration line similar to function arguments where the types and bit-widths of each signal are specified along with its name. A module body includes the internal wire and state declarations, and the statements which specify the functionality of the module. SyReC statements include reversible signal assignments, conditionals, loops, and calls of other modules.

- **SyReC Synthesis** uses the building blocks for statements and expressions as introduced in Section 3.2 to automatically synthesise reversible circuits specified in SyReC. More precisely, the following two steps are performed for each statement:

  1. Compose a sub-circuit $G_{\odot}$ realising all the expressions in a statement using the respective building blocks. The result of the expressions is buffered using additional circuit lines.
  2. Compose a sub-circuit $G_{\oplus}$ realising the overall statement using the existing building blocks of the statement together with the buffered results of the expressions.

- **Optimizing circuit synthesis** is proposed to improve the outcomes of SyReC synthesis to obtain circuits with fewer lines or lower gate costs whenever possible. **Line-aware synthesis**, as introduced in Section 3.3.1, is proposed to tackle the major drawback associated with SyReC synthesis, i.e., the high numbers of constant lines applied to the circuits. The general idea is to extend the two steps above with a third step that inversely computes the expression. So, the lines used in computing the expression are constant-valued, and hence reusable for further computations in the circuit. On the other hand, **Cost-aware synthesis** is proposed to compromise some costly gate patterns with lower-cost equivalents by using a helper-line (a trade-off technique). This one extra line is negligible and accepted to have a tangible reduction in the overall circuit cost as a trade-off.

Chapter 4

Line-aware SyReC Programming Style

Line-aware synthesis, achieves significant improvement in tackling the major drawback associated with SyReC, i.e., constant inputs [24]. This configuration reduces the constant inputs that are required to realise non-reversible operations by a cost-for-line trade-off since it duplicates the cost to make constant inputs reusable (see Section 3.3.1). However, these constant inputs are often still more than the known theoretical bounds (see Section 2.2.2).

In this chapter, we approach the problem from a different angle by which we identify a description style that can be synthesised into reversible circuits with less constant inputs. Therefore, we investigate the statements that cause these constant lines, and then we replace them, when possible, with equivalent statements that cause no or fewer lines. Based on our observations, we propose guideline rules for a line-aware programming style in SyReC, allowing for more efficient SyReC synthesis with respect to circuit lines. The influence of the proposed programming style on cost has been also considered because the expected trade-off between the reduction of lines and cost increases. This consideration highlights possible potentials to avoid or minimize cost increments.

Circuit lines with constant input is a well-known drawback associated with all hierarchical reversible synthesis methodologies. Line-awareness (in this context) means, applying less constant inputs to the circuit to compute the desired function. In SyReC, constant inputs are categorised, depending on the type of signals assigned to them, into the following:

1. `out` signals: A type of the `in/out` signal declaration of the module. Such signals can not be manipulated, because they define the external behaviour (functionality) of the system.

2. `wire` signals: Internally used signals, which facilitate computations within the module, and, hence, line-efficient programmers are supposed to declare wires when necessary with awareness to the constant inputs.

3. Implicit lines: Inevitable lines realizing irreversible computations, e.g., with binary expressions. With each binary operator, we should expect a constant-line applied to the circuit. Implicit lines are transparent to the programmer, but accumulate within the circuit and represent the main source of constant inputs among the three categories.

SyReC with line-aware synthesis configuration succeeded to moderate the accumulation of constant inputs by re-computing expressions (see section 3.3.1), which is considered for this chapter. Here, the number of lines in a circuit is bounded by the statement that requires the largest number constant lines. In general, larger expressions lead to more intermediate results to be buffered. Thus, if additional smaller statements can represent the same functionality, a further reduction in the number of lines is possible.

Example 18. *Consider the following statement:* `a += ((b & c) + ((d * e) - f))`. *To execute the outer expression (i.e., the addition operation), the intermediate results of the inner expressions* `(b & c)`, `(d * e)`, *and* `((d * e) - f)` *are buffered at the same time. This requires four circuit lines with constant inputs to buffer the results of these operations. In contrast, the same functionality can also be specified by the following statements.*

```
a += (b & c);
a += (d * e);
a -= f;
```

Here, the respective binary operations are applied separately with an assignment operation. So, no more than one ancilla line is needed to buffer the intermediate results providing reduction by 75%.

4.1 Guidelines for Line-aware Statements

4.1.1 Operator Equivalence

Realising a reversible operation does not require a constant input, by definition, while a constant input is inevitable to realise non-reversible operations. Reversible signal-assignments (`^=, +=, -=`), for instance, are mathematically equivalent to the non-reversible binary operators (`^, +, -`), respectively, e.g., `s ^= E` and `(s ^ E)` both perform the same computation. However, the circuit realisations of these equivalent computations incorporate two differences:

1. A reversible assignment does not require a constant line, while its non-reversible binary equivalent requires one to compute the expression, i.e., the assignment is more line efficient.

2. Circuits realising reversible-assignment operations have a lower cost compared to its binary operator equivalent (see Table 4.1).

An assignment statement is equivalent to two statements with shorter right-hand side (RHS) expressions if the assignment operator is equivalent to the top-level operator in the RHS expression, e.g., `s += (a + b)` is equivalent to `s += a` followed by `s += b`. This equivalence is valid because the operations (`+, -, ^`) are associative, which means the target signal `s` will be updated as desired in both cases. In splitting the expression, we replace the non-reversible operation `^` with the reversible `^=` avoiding the constant input

Table 4.1: Cost-metrics for SyReC circuits of defined operators.

Bit width	1	2	4	8	16	32	64
Operation	Gate Count						
^=	1	2	4	8	16	32	64
^	2	4	8	16	32	64	128
+= -=	1	6	20	48	104	216	440
+ -	2	8	24	56	120	248	506
Operation	Quantum Cost						
^=	1	2	4	8	16	32	64
^	2	4	8	16	32	64	128
+= -=	1	14	44	104	224	464	944
+ -	2	16	48	112	240	496	1008
Operation	Transistor Cost						
^=	8	16	32	64	128	256	512
^	16	32	64	128	256	512	1024
+= -=	8	64	208	496	1072	2224	4528
+ -	16	80	240	560	1200	2580	5040

Table 4.2: Rules of equivalence in signal-assignment statements.

Rule	Original Statement	Shorter Equivalent
1.	`s ^= (E_l ^ E_r)`	`s ^= E_l` `s ^= E_r`
2.	`s += (E_l + E_r)`	`s += E_l` `s += E_r`
3.	`s += (E_l - E_r)`	`s += E_l` `s -= E_r`
4.	`s -= (E_l + E_r)`	`s -= E_l` `s += E_r`
5.	`s -= (E_l - E_r)`	`s -= E_l` `s += E_r`
	only when (s = 0)	
6.	`s += E`	`s ^= E`
7.	`s ^= (E_l + E_r)`	`s ^= E_l` `s += E_r`
8.	`s ^= (E_l - E_r)`	`s ^= E_l` `s -= E_r`

and reducing the cost. Such assignments are enumerated, in Table 4.2, where `s` is a signal declared in the module (see rules (1–5)), `E_l` and `E_r` are the left and right operands of the RHS expression `E`, respectively. Moreover, when `s` is known to be constant $'0'$ (e.g., a non-initialized `out` or `wire` signal), then `(0 + E) = (0 ^ E) = E`. For the same reason += and ^= are equivalents when `(s = 0)` , i.e., rule 6 holds meaning rules 7 and 8 hold as well.

The two operands `E_l` and `E_r` are grammatically defined to be expressions in the general case. Here, the major advantage of splitting an expression into two shorter expressions results in less constant lines if this expression is the largest in the module.

Example 19. *The SyReC module in Figure 4.1(a) contains three statements. The largest expression appears in line 4, and computed using* ***6*** *operators. Consequently,* ***6*** *constant inputs (each using an 8-bit bundle, i.e., 8-circuit lines) are implicitly added to the circuit to realise it. The binary expression has a left operand* `((a * b) - (a / b))` *(****3*** *operators), a right operand* `((a + b) / t)` *(****2*** *operators), and a top-level operator* `(+)`*. The statement can be replaced with:*

```
x += ((a * b) - (a / b))
x += ((a + b) / t))
```

or, even shorter, with:

```
x += (a * b)
x -= (a / b)
x += ((a + b) / t))
```

to result in the code in Figure 4.1(b).

```
1  module example(in a(8), in b(8), inout x(8), out f(8))
2  wire t(8)
3  t ^= (a & b)
4  x += (((a * b) - (a / b)) + ((a + b) / t))
5  f ^= (((t ^ b) + x) * (a - b))
```

(a) Simple SyReC module

```
1  module example(in a(8), in b(8), inout x(8), out f(8))
2  wire t(8)
3  t ^= (a & b)
4  x += (a * b)
5  x -= (a / b)
6  x += ((a + b) / t)
7  f ^= (((t ^ b) + x) * (a - b))
```

(b) Equivalent module using shorter statements

Figure 4.1: Two equivalent SyReC modules.

Despite the equivalence of codes in Figure 4.1(a) and Figure 4.1(b), line-aware SyReC synthesis exploits fewer lines to realise the latter (see Table 4.3). This table shows that the second code realises the circuit using fewer lines with a lower cost.

4.1.2 Internal Wires

Internal wires are usually declared to avoid repeating a computation. For example, the expression `(a & b)` is assigned to `wire t` (Figure 4.1(b) line 3), instead of repeating this computation twice in Lines 6 and 7. Here, we avoid duplicating circuitry to avoid additional gate cost. We reduce the size of these expressions by substituting the duplicated computation by a signal identifier, which means less constant lines to realise the expression. This motivates wire declaration with the intention to reduce the size of the largest expression in the module.

Example 20. *The largest expression in Figure 4.1(b) appears in the RHS of line 7 and is computed using* ***4*** *operators. The top-level operation is (∗), which has no reversible equivalent, so expression split, as explained in Section 4.1.1, is not applicable. However, this problem can be circumvented if a wire* `w` *is declared (Figure 4.2, line 2), which applies a constant signal to the circuit and allows for the copy of the operand* `((t + b) ^ x)`*, as shown in Figure 4.2 line 7. Then,* `w` *substitutes the operand in the expression (Figure 4.2, line 8). This code is described using expressions computed with* ***2****, or less, operators each.*

```
1        module example(in a(8), in b(8), inout x(8), out f(8))
2        wire t(8),w(8)
3        t ^= (a & b)
4        x += (a * b)
5        x -= (a / b)
6        x += ((a + b) / t)
7        w ^= ((t ^ b) + x)
8        f ^= ( w * (a - b))
```

Figure 4.2: Simple SyReC program with extra wire.

Although a wire also requires its "own" constant input, this code is realised with less circuit lines (see Table 4.3). Unlike the expression split proposed in Section 4.1.1, which is advantageous with respect to all circuit parameters, applying a wire can be advantageous only when used properly. A slight increment in the circuit cost is expected due to the additional `^=` operator to assign the expression to the wire along with the extra constant input due to the new wire. So, we have the intended advantage of declaring a wire when:

1. it is applied to the largest expression only.

2. it is assigned by the larger among the two operands. In this case the top-level operation will be computed with the shorter expression as we observe better results when the two operands are of almost similar sizes.

3. the assigned operand contains at least two operators.

The possibility of another split sometimes appears after the new wire assignment statement, e.g., line 7 in Figure 4.2, can be replaced, as shown in Figure 4.3. Here, the split does not reduce the number of circuit lines because the largest expression in the module remains with the same number of operations, and it shows a slight improvement in cost metrics (see Table 4.3).

```
1   module example(in a(8), in b(8), inout x(8), out f(8))
2   wire t(8),w(8)
3   t ^= (a & b)
4   x += (a * b)
5   x -= (a / b)
6   x += ((a + b) / t)
7   w ^= t
8   w ^= b
9   w += x
10  f ^= ( w * (a - b))
```

Figure 4.3: Simple SyReC program with extra wire.

Removing a wire means reducing a constant input. Therefore we should seek for redundant wires that can be removed without loosing the advantage of using them. One such case can be identified in Figure 4.3, line 7. We assign (copy) the value of the wire `t` to the constant $'0'$ wire `w`. Then, `t` is considered as a **garbage** output. We recognise that `w` is redundant, because it can be simply removed and substituted by `t` (see Figure 4.4).

```
1   module example(in a(8), in b(8), inout x(8), out f(8))
2   wire t(8)
3   t ^= (a & b)
4   x += (a * b)
5   x -= (a / b)
6   x += ((a + b) / t)
7   t ^= b
8   t += x
9   f ^= ( t * (a - b))
```

Figure 4.4: Simple SyReC program with wire removed.

4.1.3 Temporary Signal Update

The tendency to describe the desired function with shorter expressions may reach a limit where further reduction is no more possible. In Figure 4.4 two expressions appear with **two** operations each (lines 6 and 9), but neither can be reduced as discussed in Sections 4.1.1 and 4.1.2. In both expressions, we could identify an operator that has a reversible-assignment equivalence `(+,-,^)`, but not as a top-level operation, e.g., in line 9 with `f ^= (t * (a - b))`. Here, we can update the signal `a` before this statement and use the updated value to compute the expression, `a -= b; f ^= (t * a)`. We must be aware that this is only possible because signal `a` (1) is not used again in this expression and

(2) garbage after this statement (an `in` or `wire` signal that is not used after the statement within the module).

If the updated signal is not garbage, then the update should be only temporary and later inversely-updated directly. This inverse update of an increase operation, +=, is decrease, −, and vice versa, while the XOR-update, ^=, is self-inverse.

```
1   module example(in a(8), in b(8), inout x(8), out f(8))
2   wire t(8)
3   t ^= (a & b)
4   x += (a * b)
5   x -= (a / b)
6   a+= b
7   x += (a / t)
8   a-= b
9   t ^= b
10  t += x
11  a -= b
12  f ^= (t * a)
```

Figure 4.5: Simple SyReC program with a temporary signal update.

Example 21. *In Figure 4.5, signal* `a` *is temporarily updated in line 6 and before being used in line 7 to compute signal* `x`*. In line 8,* `a` *is inversely updated to its original value because it is not yet garbage. Another temporary update for* `a` *is made in line 11, and then used to compute* `f` *in line 12. Here, no inverse update is necessary, because* `a` *is subsequently garbage. This temporary signal update allows the description of the desired function using very short expressions with only* ***1*** *or even no operator at all.*

Table 4.3: Circuit metrics for equivalent SyReC modules.

Metric	Fig.4.1(a)	Fig.4.1(b)	Fig.4.2	Fig.4.3	Fig.4.4	Fig.4.5
Circuit Lines	88	72	64	64	56	48
Circuit Lines (8-bit)	11	9	8	8	7	6
explicit wires (8-bit)	1	1	2	2	1	1
Implicit lines (8-bit)	6	4	2	2	2	1
Gate Count	3756	3628	3636	3548	3540	3460
Quantum Cost	69064	68824	68832	68688	68680	68544
Transistor Cost	92944	91696	91760	90944	90880	90128

The inverse computation statements inserted in the code are, practically, extra re-compute circuits (cost) described explicitly in the code. In other words, they are cost-for-lines trade-offs. In this example, the cost increment is compensated with the cost reduction due to the replacement of the binary operators `(+, -)` by the assignment operators `(+=, -=)`, respectively (see Table 4.3).

In the same way, internal wire signals can be inversely computed to their original constant $'0'$ value, and with this, the wires become reusable for further computation instead of declaring different wires.

Table 4.4: Experimental evaluation.

Benchmark	L_{io} lines	Total Lines L_t (1)	(2)	(3)	Gate Count G (1)	(2)	(3)	Quantum Cost Q (1)	(2)	(3)	Transistor Cost T (1)	(2)	(3)
factorization_4th_order	48	152	104	72	873	1704	1116	4841	9640	6956	13568	26752	18592
alu_flat	50	118	67	67	3671	7286	7014	181662	363012	361172	179464	357904	353072
cpu_alu_16bit	55	404	142	142	6237	12591	12099	662531	1281717	1229192	568328	1103200	1058584
factorization_8th_order	80	280	168	120	1729	3392	2508	9665	19264	16060	27008	53376	42464
variance_10	96	376	208	128	4320	7620	6900	44626	87168	85888	79088	147920	141040
varops	96	224	192	192	1305	1928	1368	2801	4176	3120	13424	19920	14448
lu	98	197	133	132	384	612	516	6557	10462	7966	9856	15616	12544
simple_alu	98	229	133	133	3978	7832	7172	144791	287346	278504	154432	305536	290272
alu	98	229	133	133	14459	28794	28232	1704912	3407588	3401244	1402232	2801136	278844
cpu_alu_32bit	103	756	254	252	20381	40975	39708	2235491	4381653	4222827	1917448	3766496	3635136
acceleration	160	352	288	256	24908	49752	49624	2435006	4869948	4869820	2058576	4116640	4115616
cpu_control_unit	232	391	290	272	1243	2588	2415	40433	80142	79093	43888	87176	84472

(1): Measured for circuits realised with original SyReC synthesis as described in Section 3.2.
(2): Measured for circuits realised with line-aware SyReC synthesis as described in Section 3.3.1.
(3): Measured for circuits specified with rewritten codes as proposed in this chapter.

4.2 Experimental Evaluation

Despite their equivalence, SyReC codes shown in Figures 4.1 through 4.5 have different circuit realisations of the desired function. Table 4.3 shows the impact of applying the ideas discussed in Sections 4.1.1, 4.1.2, and 4.1.3 on the resulting circuits' parameters, i.e., the number of lines and cost measures. When we compare the spontaneously written code (Figure 4.1) with the code written according to the proposed programming style (Figures 4.5), we can see a dramatic reduction in the number of constant inputs. Moreover, this reduction is not a trade-off as circuit cost measures show a slight reduction. The other important observation related to these two codes is that the proposed style describes the module with a large number of shorter statements, as compared to the spontaneous code, and is less readable.

The function described in these codes has no practical meaning. It has been designed to demonstrate the ideas with a simple example. A valid experimental evaluation, on the other hand, was carried out on a true set of benchmarks describing meaningful functions. Each benchmark is realised with three different configurations:

(1) The original SyReC synthesis without any optimisation as described in Section 3.2.

(2) SyReC with line-aware synthesis as described in Section 3.3.1.

(3) The proposed programming style is applied to rewrite the benchmark and then realised using SyReC with line-aware synthesis.

These configurations result in three different circuits for each benchmark. Four parameters are measured to characterise each reversible circuit: number of lines (L_t), gate count (D), quantum cost (Q), and transistor cost (T) metrics. In addition to these measures, the number of lines declared for `in` and `out` signals is computed as L_{io} (see Table 4.4).

4.2.1 Normalised Circuit Metrics

The benchmarks used in evaluating the proposed programming style are significantly different in many features including the sizes. Hence, we need to normalise circuit metrics to have a better general assessment of the proposed style that is independent of the problem size. Here, we have two key metrics:

1. The impact on the circuit lines is supposed to consider the number of constant lines added to realise the function, which is the primary drawback of this approach. In theory, a pure reversible computation is performed within the lines declared for the input and output signals. Therefore, our normalisation excludes these lines and uses them as a reference to define the problem size, with respect to the factor of circuit lines as follows:

$$Ln_x\% = \frac{L_{t_x} - L_{io}}{L_{io}} \times 100\% \tag{4.1}$$

2. The impact on circuit cost (complexity) is normalised with respect to the original realisation scenario as follows:

$$Cn_x\% = \frac{C_x - C_1}{C_1} \times 100\% \tag{4.2}$$

where, C can take any cost metric, i.e., D, Q or T, while the subscript $x = 2$ or 3 refers to the realisation configuration in the table.

The normalisation results for the considered configurations ((2) and (3)) are shown in Figures 4.6, 4.7, 4.8, and 4.9, for normalized lines, gates, and quantum and transistor cost metrics, respectively. In each graph, we recognize pairs of bars. Darker bars refer to the spontaneously programmed benchmarks and lighter bars refer to the rewritten benchmarks all realised using SyReC with line-aware synthesis.

4.2.2 Discussion of Results

The normalisation as discussed in Section 4.2.1 simplifies the assessment of the proposed programming style on the realisation of the twelve benchmarks. It shows different impacts on various benchmarks where we compute the improvement by the difference between the two normalised measures of each benchmark in each of these four figures, i.e.,

$$D_{Ln}\% = Ln_2 - Ln_3 = \frac{L_{t_2} - L_{t_3}}{L_{io}} \times 100\% \tag{4.3}$$

$$D_{Cn}\% = Cn_2 - Cn_3 = \frac{C_2 - C_3}{C_1} \times 100\% \tag{4.4}$$

Accordingly, we may categorise the improvements achieved by rewriting the benchmarks as proposed to be:

1. **significant**, when the improvement is more than **20%**.
2. **tangible (noticeable)**, when the improvement is between **5%–20%**.
3. **insignificant**, when the improvement is less than **5%**.
4. **negative**, when the result is worst in the proposed style. This can occur because of a cost-for-line trade-off.

Then the results show:

1. **2/12** benchmarks offer **significant** improvement in **all** circuit parameters.
2. **4/12** benchmarks offer **significant** improvement in at **least** one circuit parameter.
3. **5/12** benchmarks offer **tangible** improvement in at least **one** circuit parameter.
4. **1/12** benchmark offers an **insignificant** improvement in all circuit parameters.

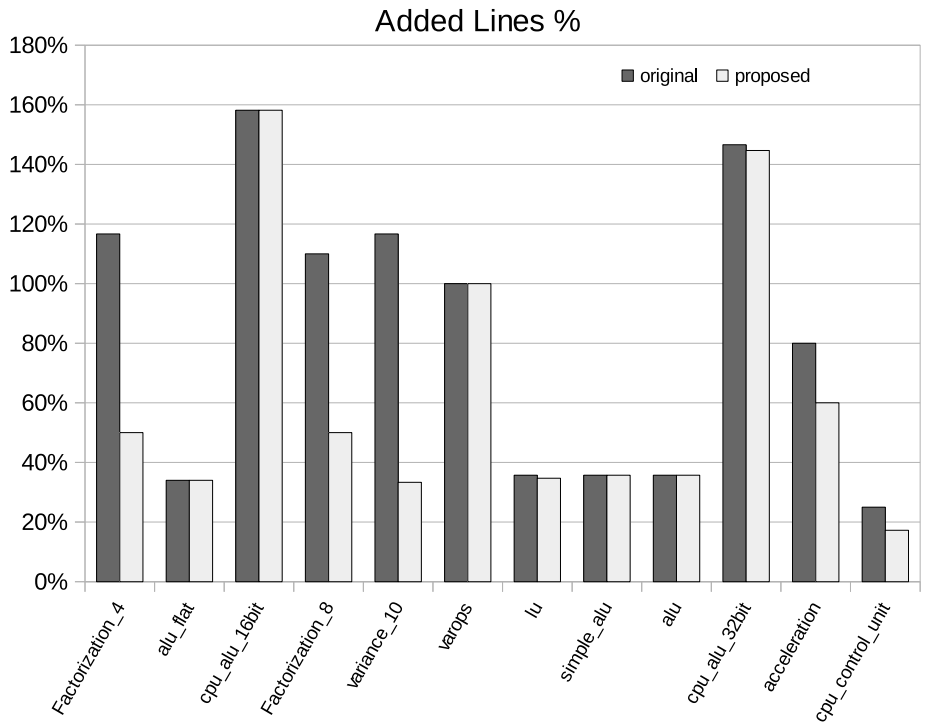


Figure 4.6: Normalised increase in constant $'0'$ circuit lines.

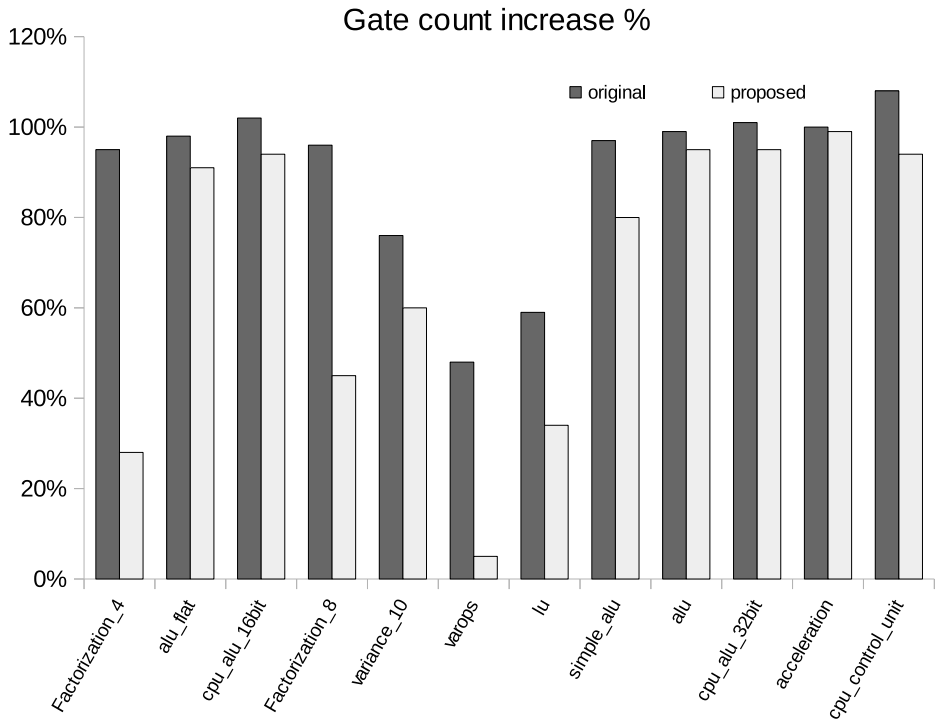


Figure 4.7: Normalised increase in gate count.

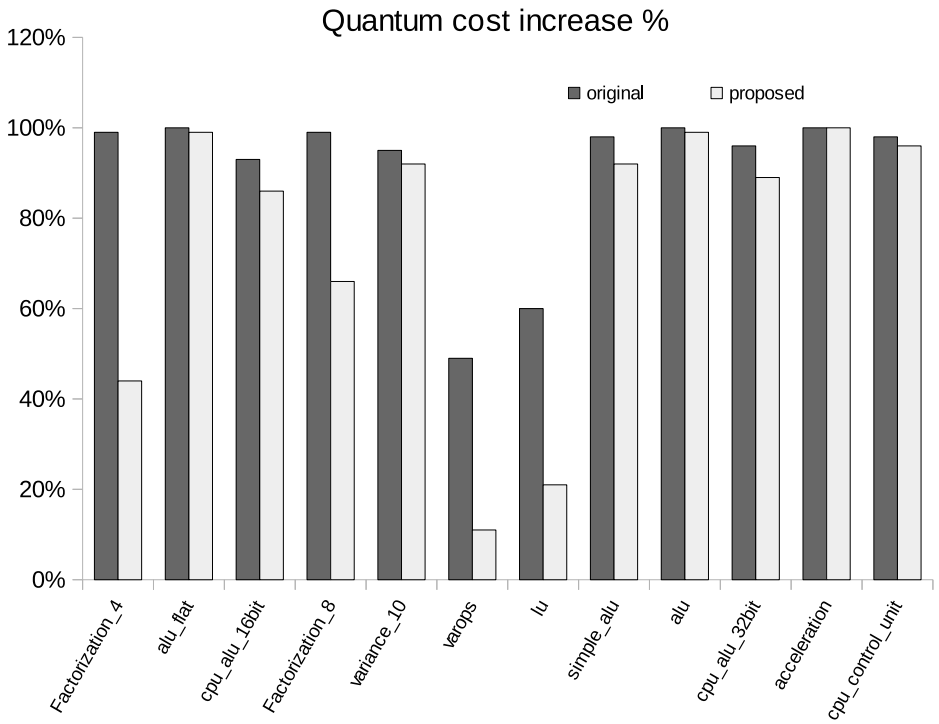


Figure 4.8: Normalised increase in quantum cost.

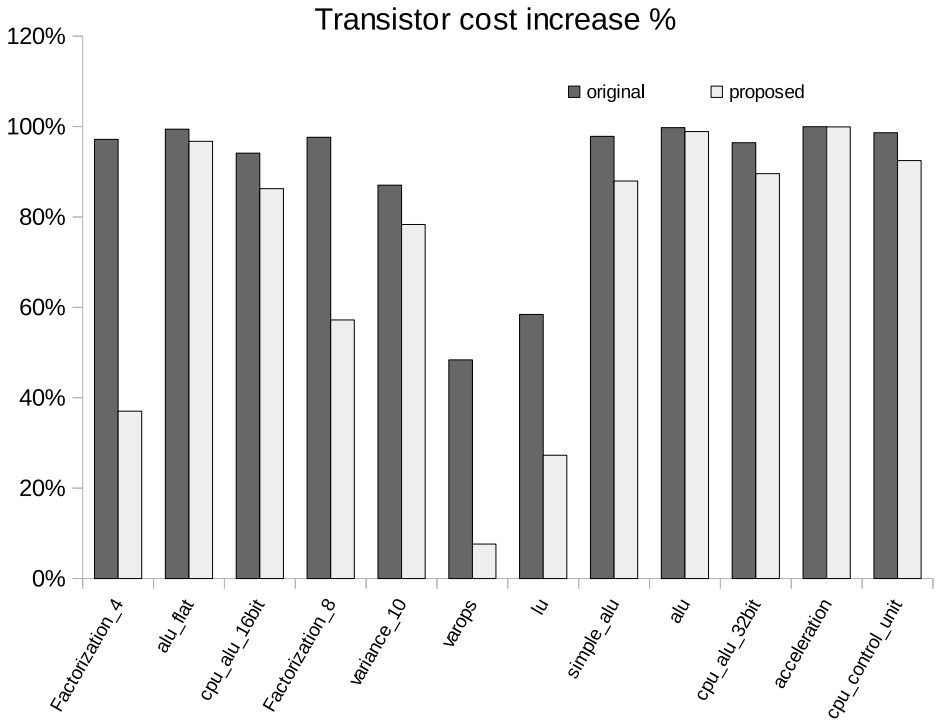


Figure 4.9: Normalised increase in transistor cost.

4.3 Summary

In this chapter, we proposed a SyReC programming style oriented towards optimised circuit synthesis. The primary optimisation objective is to reduce the number of constant inputs for which SyReC with line-aware synthesis is considered. The secondary objective is to avoid, or at least to reduce, the cost-for-line trade-off associated with line-reduction. The style is based on describing the desired function with as short expressions as possible guided by the following:

1. Splitting assignment statements by replacing some non-reversible binary operators by their equivalent reversible assignment operators (Section 4.1.1).

2. Substituting the operands of large expressions by an internal wire identifier (Section 4.1.2).

3. Temporary signal update, with inverse-update when necessary (Section 4.1.3).

An experimental evaluation carried out to compare the spontaneous and the proposed SyReC programming styles with results showing a significant improvement, in one or both optimisation objectives, in the clear majority of these SyReC benchmarks (Section 4.2).

The proposed style shows code that is less readable, as compared to the spontaneously written code. The modules are described with sorter expressions and with a large number of statements. The most interesting observation in this experiment is that the reduction in the number of lines is not accompanied with extra cost. In fact, the partial increase is compensated within the circuit, in all benchmarks.

Overall, the evaluation shows that re-writing the SyReC description for reversible circuit synthesis is a promising direction to obtain compact results compared to the current state-of-the-art.

An interesting observation is that despite the theoretical possibility, none of these benchmarks shows a negative impact on any circuit parameter as a cost for line trade-off. These trade-offs where partial and are compensated within the circuit in all benchmarks. This observation shows that re-writing SyReC descriptions for reversible circuit synthesis is a promising direction to obtain more efficient tool for HDL-based synthesis.

Chapter 5

Optimised Synthesis of SyReC Expressions

The line-aware SyReC programming style proposed in Chapter 4 tackles the problem of constant inputs within the source code before working with the synthesis phase. This arrangement results in better circuits, but it compromises spontaneous HDL programming and the code readability. Even with a fully automated code conversion, human intervention will be expected for further tasks, such as debugging, which will be more complicated with less readable code. This contradicts a basic HDL characteristic, which is the simplicity of description, especially when handling complex problems. In this chapter, we analyse the problem at another level of abstraction, which is transparent to the programmer, to achieve line-awareness even with a spontaneous programming style. As we have already identified, SyReC expressions are the main source of constant inputs in the circuit. Hence, if line-awareness, as an objective, is considered when realising expressions, it may solve the problem, at least to some extent, without the need to change the style of programming in SyReC.

It is important not to be confused with the line-aware statement synthesis configuration introduced in Section 3.3, where expressions are computed in the same way and then re-computed to free the garbage lines. Here, we propose a procedure to compute SyReC expressions with line-awareness (i.e., with fewer lines).

In the first section of this chapter, computation of SyReC expressions are reviewed with more details as compared to the brief review in Section 3.2.2. The modified line-aware synthesis algorithm is proposed in Section 5.2. Circuit cost is considered in Section 5.3 along with other related issues discussed before the experimental evaluation is introduced in Section 5.5.

5.1 SyReC Expressions

An expression, as defined in SyReC grammar, can be a numeral, a signal identifier or an operation on some operand(s). The latter is a combined structure of an operator along with its operand(s) has to be computed (see Table 3.3). Operands, in the general case, are also defined to be an expression. This recursive definition allows one expression to describe complex computations. SyReC defines a reversible circuit to compute each operation, so operations are considered basic building blocks (the lowest level in the synthesis hierarchy). These circuits were introduced in detail in [63]. In this chapter, a single operation is considered a unit of computation, i.e., the complexity of an expression is measured as the number of operations performed to compute it. Table 3.3 shows two types of SyReC operators:

1. Binary operators $\odot$ with two operands E_L and E_R. A binary expression is defined with mandatory parentheses, i.e., $(E_L \odot E_R)$.
2. Unary operators $\ominus$ with only one operand E_U directly to the right of the operator, i.e., $\ominus E_U$. There are only two such operators in SyReC (`~` and `!`).

An expression can be computed only when its operand(s) are already computed. This gives a tree structure to the expression where a node represents an operation, and an edge is a reference to an operand. The symbol of an operation is written inside the node. Figure 5.1 provides symbolic representations of these nodes where

(a) a node of a binary operation has two edges, one black edge and another grey edge, referring to its left and right operands, respectively.

(b) a node of a unary operation has only one black edge referring to its operand.

(a) Binary (b) Unary

Figure 5.1: Tree representation of SyReC operations.

Example 22. *The SyReC expression* `(((a * b) / ~(c + a)) - ~(b * c))` *is computed using seven operations, and Figure 5.2 shows the tree representation of this expression. The top-level operation in the expression (subtraction* `(-)`*) is the root node of the tree. Subtraction is a binary operation with two operands, and both are expressions. The left operand is* `((a * b) / ~(c + a))` *and the right operand is* `~(b * c)`*, each of which is a sub-tree in the structure. The expression should be parsed and recursively computed using the same steps for each sub-tree until reaching the primary signals at the tree bottom.*

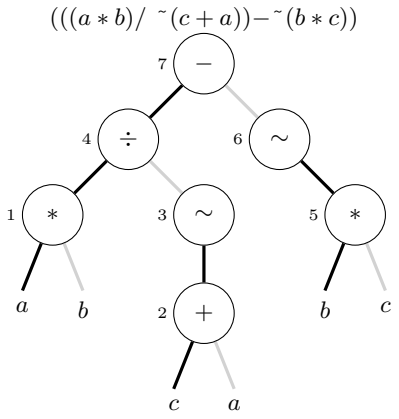
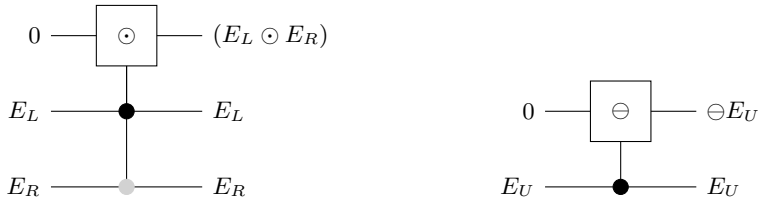


Figure 5.2: Expression tree for Example 22.

As shown in Figure 5.2, the precedence of an operation (the order of computation) is tagged to the left of the node as an integer, i.e., the node tagged 1 is computed first, so the top-level node is tagged with the highest number, i.e., 7 in this expression. The tree-bottom shows primary values, such as signal identifiers or numbers, which require no computation. In other words, each operator has lower precedence as compared to those operators used in computing its operand expression(s).

5.1.1 Circuit Realisation of SyReC Operations

SyReC is not restricted to reversible operations, as shown in Table 3.3. The majority of operations defined by SyReC are irreversible. Consequently, a constant input is applied to the circuit to compute the operation.



(a) Binary Operation (b) Unary Operation

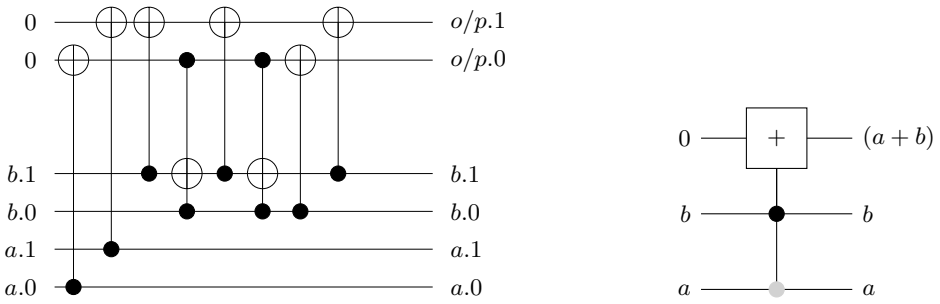
Figure 5.3: Schematic representation of SyReC operations.

Figure 5.3 shows the symbolic representation of a SyReC operation in a schematic diagram, such that:

(a) A circuit of a binary operation $G_{\odot}$ is computed to a target line with constant $'0'$ input. The two operands are connected from two inputs serving as control lines in the reversible circuit. The left operand E_L is identified by a black dot, while a grey dot identify the right operand E_R. At the output side of the circuit the expression, $(E_L \odot E_R)$ is computed to the target line while the two operands' lines are unchanged.

(b) A circuit of a unary operation $G_{\ominus}$ is also computed to a line with constant $'0'$ input. Here, only one input is serving as a control line in the reversible circuit. A black dot identifies the operand E_U. At the output side of the circuit, the expression $\ominus E_U$ is computed to the target line while the operand line is unchanged.

This schematic representation keeps the discussion at a high-level of abstraction by hiding the gate-level details, such as the bit-widths influence on the circuit details. The level of discussion is comparable to register-transfer level in the design flow of conventional circuits, e.g., Figure 5.4 shows a 2-bit addition expression `(a + b)`. The circuit on the gate level in Figure 5.4(a) shows more details as compared to the schematic representation in Figure 5.4(b). In this chapter, circuit realisation and synthesis is maintained at this level without being involved in the circuit details, especially those related with the bit-width. Consequently, each operation is considered a unit building block in the expression circuit, and its bit-width is considered a single line for the remainder of this chapter.



(a) Gate-level (b) Schematic

Figure 5.4: SyReC defined 2-bit addition $G_{(a+b)}$.

5.1.2 SyReC Synthesis of Combined Expressions

The discussion in Section 5.1.1 assumes the operands are already available, and ignores the fact that these operands require computation if they are expressions. SyReC does not provide a ready circuit definition for combined expressions (those with more than one operation), but synthesise such expression as a sequence of cascading operation blocks.

For a combined binary expression $(E_L \odot E_R)$, the circuit $G_{(E_L \odot E_R)}$ is realised by cascading three parts as described in Equation 5.1:

$$G_{(E_L \odot E_R)} = G_{E_L} \;\; G_{E_R} \;\; G_{\odot} \tag{5.1}$$

where G_{E_L} and G_{E_R} are the circuits to compute the left and right operands E_L and E_R, respectively, and $G_{\odot}$ is the defined circuit to compute the top-level binary operation $\odot$. On the other hand, only two parts are required to synthesise the circuit $G_{\ominus E_U}$ of a unary expression, $\ominus E_U$. First, the operand E_U is computed using the circuit G_{E_U}, then it is cascaded by the operation circuit $G_{\ominus}$, as described in Equation 5.2:

$$G_{\ominus E_U} = G_{E_U} \;\; G_{\ominus} \tag{5.2}$$

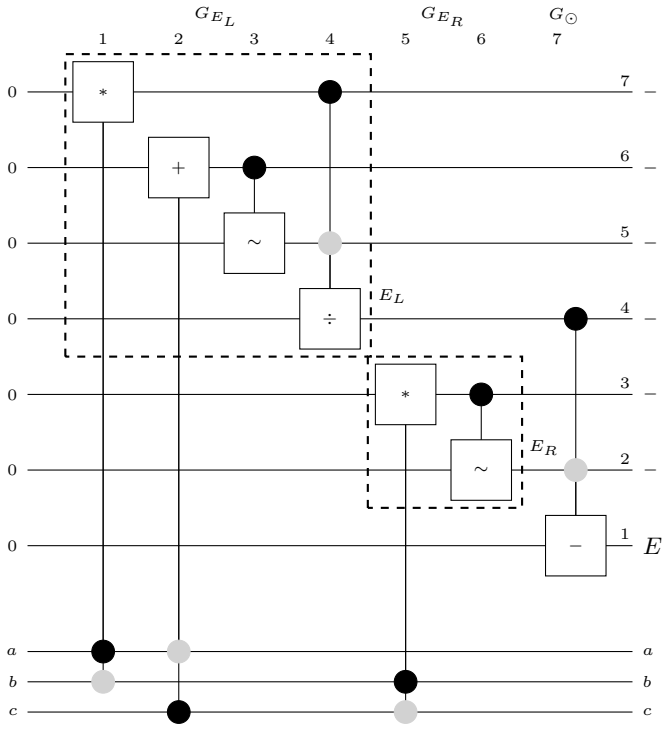


Figure 5.5: Block diagram for SyReC synthesis of the expression in Example 22.

This recursive synthesis of operand computation continues until reaching a signal identifier or a number, which does not need any computation. In other words no circuit is needed to compute this operand. This is the base case for the recursion.

In practice, SyReC parses the expressions into tree structures and traverses these trees to identify the precedence of operations to determine the order by which circuits are cascaded.

Example 23. *The expression* `(((a * b) / ~(c + a)) - ~(b * c))` *in Example 22 is computed as shown in Figure5.5. The circuit cascade follows the operation precedence as specified in Figure 5.2, which begins by computing the operation tagged with number* 1*, i.e.,* `(a * b)`*, and so on until the top top-level subtraction operation* − *is computed at the final stage of the cascade.*

5.1.3 Constant Inputs

A constant $'0'$ line is applied to the input of the circuit for each operation used to compute the expression in Figure 5.5. This yields a circuit G_E that computes an expression E realized with k constant $'0'$ lines applied to its input when E is computed using k operations, as will be explained in next. Figure 5.6 shows the schematic symbol for a generic expression circuit, G_E.

In more detail, Figure 5.7 shows the generic schematic diagram to realise (a) a binary expression, $G_{(E_L \odot E_R)}$, and (b) a unary expression, $G_{\ominus E_U}$, which both show how constant $'0'$ lines are accumulated to be $(\lambda_E = k)$ as follows:

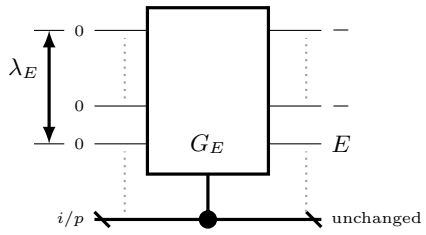


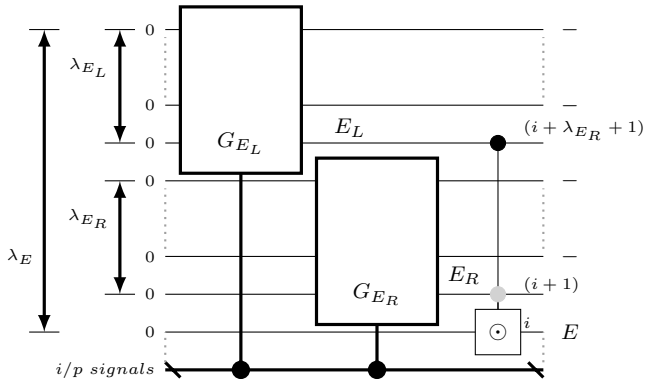
Figure 5.6: Generic schematic representation of a combined expression circuit G_E.

1. for a binary-expression $G_{(E_L \odot E_R)}$:

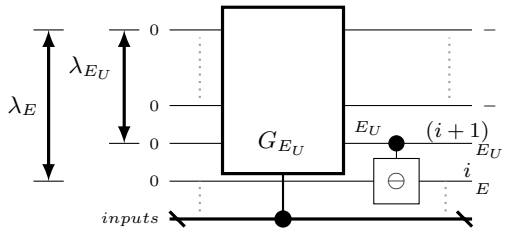
$$\lambda_E = \lambda_{E_L} + \lambda_{E_R} + 1 = k \tag{5.3}$$

2. for a unary-expression $G_{\ominus E_U}$:

$$\lambda_E = \lambda_{E_U} + 1 = k \tag{5.4}$$



(a) Binary-Expression $G_{(E_L \odot E_R)}$



(b) Unary-Expression $G_{\ominus E_U}$

Figure 5.7: Generic schematic diagram to synthesise SyReC expressions G_E.

where k is the number of operators in the expression E, and λ_x is the number of constant $'0'$ lines applied to realise G_x. The recursive nature of these two equations requires that to calculate λ_E, we need to apply the same equations to calculate λ_{E_L}, λ_{E_R}, or λ_{E_U}, until reaching a base case of recursion (a primary input).

Example 24. *To calculate the number of lines $\lambda_E$for the binary expression in Example 23,* `(((a * b) / ~(c + a)) - ~(b * c))`*, we apply Equation 5.3. Here, E_L is* `((a * b) / ~(c + a))` *and E_R is* `~(b * c)`*, which must be subjected the same calculations. E_L is also a binary expression with a left operand* `(a * b)` *and a right operand* `~(c + a)`*. Here, $\lambda_{(a*b)=1}$ because it has two signal identifiers* `a` *and* `b` *(base case expressions). On the other hand, calculating the lines in the operand* `~(c + a)` *follows the formula in Equation 5.4 because this is a unary expression, i.e., $\lambda_{\sim(c+a)} = 1 + 1 = 2$. Now, we may calculate $\lambda_{E_L} = 1+2+1 = 4$. In the same way, Equation 5.4 is applied on the unary expression E_R to compute $\lambda_{E_R} = 1 + 1 = 2$. Finally, the lines for the top-level expression are calculated, i.e., $\lambda_E = 4 + 2 + 1 = 7$.*

5.2 Line-aware Synthesis

The accumulation of constant inputs when computing SyReC expressions, as discussed in Section 5.1.2, causes the critical drawback associated with this synthesis approach.

Fig 5.5 shows the desired output is only one line, while the other $(k-1)$ lines are used to compute intermediate operations, which are considered as garbage outputs. To this end, each line is processed only once to compute an operation. Then, the line, except the final output, is used only once as an operand to compute another expression. Next, the line is considered as garbage, except the output line. One possibility to realise expressions with less lines is by reusing, instead of appending, lines. To achieve this, we need to:

1. define a garbage-free circuit that computes expression E with one output line and $k-1$ lines inversely computed (re-computed) to constant $'0'$.

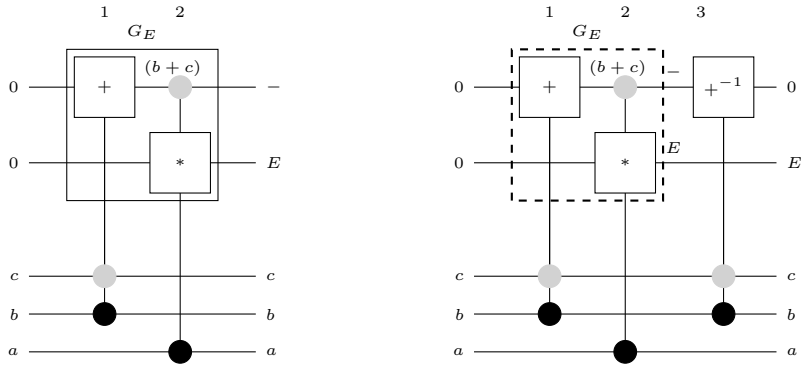
2. reuse the re-computed lines of one operand to realise the other in an expression.

5.2.1 Garbage-free Expressions

Realising an expression with no garbage is possible if the garbage lines are inversely computed to constant $'0'$, i.e., by the identity cascade ($G\ G^{-1}$). This means that circuit lines are carrying the same values at both ends, which is equivalent to a circuit with no gates. Now, if the expression circuit G_E is defined such that $G_E = G_x G_\circ$, where $G_\circ$ is the final operation to compute the top-level operation, then G_x is the circuit to compute the operand(s).

$$G_x = \begin{cases} G_{E_L} G_{E_R}, & \text{if } \circ = \odot \text{ (binary-expression).} \\ G_{E_U}, & \text{if } \circ = \ominus \text{ (unary-expression).} \end{cases} \tag{5.5}$$

We see $G_\circ$ computes the desired output, while all the lines of G_x are garbage. As long as G_x garbage outputs are not affected by computing $G_\circ$, we can append G_x^{-1}, i.e., $G_x G_\circ G_x^{-1}$ to re-compute these garbage lines.



(a) G_E with a garbage line (b) G_E' with no garbage

Figure 5.8: Two equivalent realisations for $G_{(a*(b+c)}$ in Example 25.

Example 25. `(a*(b+c))` *is an expression (E) composed using two operations. Figure 5.8a shows the circuit diagram of* G_E *with a garbage line at the output end. The circuit is synthesised by cascading two operations and required two constant* $'0'$ *lines. In this expression,* $G_x = G_+$ *is the first operation, and the second is* G_**, which calculates the final output.* $G_E = G_+\ G_*$ *has to be cascaded by* $G_x^{-1} = G_+^{-1}$ *to re-compute the garbage line, as shown in Figure 5.8b, which is the garbage-free circuit,* $G_E' = G_+\ G_*\ G_+^{-1}$*.*

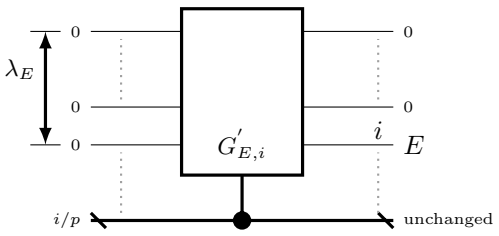


Figure 5.9: Generic representation of a garbage-free Circuit G_E'.

To generalize; we define G_E' to be the garbage-free equivalent of G_E, where all intermediate lines are re-computed to constant $'0'$ (Figure 5.9) as:

$$G_E' \ = \ G_x\ G_\circ\ G_x^{-1} \ = \ G_E\ G_x^{-1} \tag{5.6}$$

This circuit cascade is schematically represented in Figure 5.10.

By substituting G_x as defined in Equations 5.5 and 5.6, we obtain the following two equations:

$$G'_{(E_L \odot E_R)} = G_{E_L}\ \ G_{E_R}\ \ G_\odot\ \ G_{E_R}^{-1}\ \ G_{E_L}^{-1} \tag{5.7}$$

$$G'_{\ominus E_U} = G_{E_U}\ \ G_\ominus\ \ G_{E_U}^{-1} \tag{5.8}$$

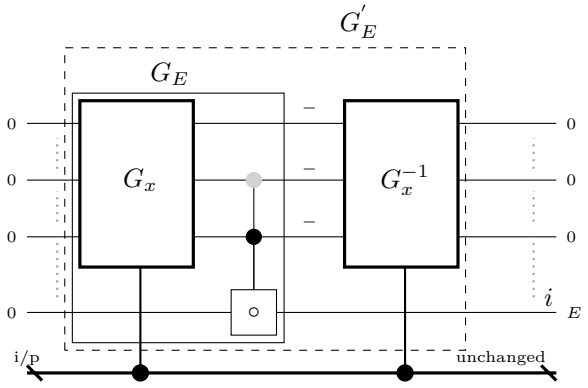


Figure 5.10: A schematic diagram for the garbage-free realisation G'_E.

With this approach, we defined garbage-free circuits that realise SyReC expressions with a higher cost, due to the extra circuitry of inverse computation G_x, but with no impact on the number of lines used. Here, we have no advantage unless this arrangement is invested in reusing lines.

5.2.2 Reusing Constant Inputs

After re-computing, lines are eligible to be used in another computation. For example, in a binary expression, $(E_L \odot E_R)$, when the left-operand E_L is computed with a garbage-free circuit G'_{E_L}, then the re-computed lines can be used in computing the right operand, G_{E_R}. This modifies the circuit definition in Equation 5.1 to be

$$G_{(E_L \odot E_R)} = G'_{E_L}\ G_{E_R}\ G_{\odot}. \tag{5.9}$$

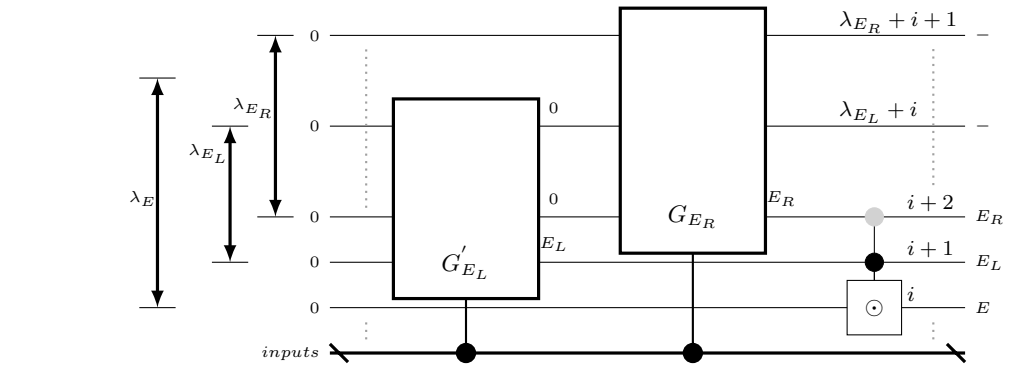
We can see the number of constant lines applied depends on the largest operand (see Figure 5.11(a)), i.e., $\lambda_E = \max\langle(\lambda_{E_L}+1), (\lambda_{E_R}+2)\rangle$ instead of the summation, i.e.,

$$\lambda_{(E_L \odot E_R)} = \begin{cases} \lambda_{E_L}+1, & \text{if } \lambda_{E_L} > \lambda_{E_R} \\ \lambda_{E_R}+2, & \text{if } \lambda_{E_L} \leqslant \lambda_{E_R} \end{cases} \tag{5.10}$$

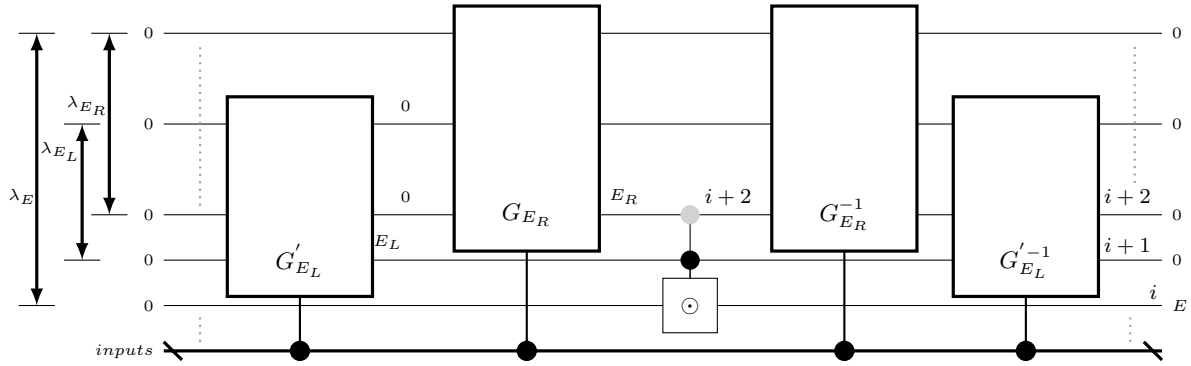
Consequently, the definition of the garbage-free equivalent of this circuit G'_E, as defined by Equation 5.7, should also be modified to incorporate this scheme (see Equation 5.11). The number of lines applied to realise G'_E is the same number of lines computed from Equation 5.10 (see Figure 5.11(b)).

$$G'_{(E_L \odot E_R)} = G'_{E_L}\ G_{E_R}\ G_{\odot}\ G^{-1}_{E_R}\ G'^{-1}_{E_L} \tag{5.11}$$

Line-ware synthesis of SyReC expressions follows the circuit cascades as described in Equations 5.9 and 5.11, as well as 5.2 and 5.8, which are mutually recursive.



(a) $G_{(E_L \odot E_R)}$



(b) $G'_{(E_L \odot E_R)}$

Figure 5.11: Schematic diagram for garbage-free line-aware binary expression synthesis.

Example 26. *Applying Equations 5.9 and 5.11 instead of Equations 5.1 and 5.7 to realise the expression in Example 22 results in a circuit with* ***5*** *constant* $'0'$ *lines (Figure 5.12), as compared to* ***7*** *in Figure 5.5. Here,* G'_{E_L} *is computed instead of* G_{E_L}*, with lines' numbers* $3, 4$*, and* 5 *being re-computed (shaded area). Then, lines* 3 *and* 4 *are reused in computing* G_{E_R} *instead of lines* 6 *and* 7 *in Figure 5.5. The total number of lines* λ_E *for this expression is calculated from Equation 5.10, such that* $\lambda_E = (4+1) = 5$*.*

5.2.3 Operands Order in Synthesis

Both original SyReC and line-aware synthesis of binary expressions as proposed in Sections 5.1.2 and 5.2.2, respectively, start with computing the left operand circuit G_{E_L} first before computing the right operand G_{E_R}. This default case is not mandatory because the operands are independent of each other. The only consideration is to guarantee that both operands are computed before the top-level operation. There is no reason to change the usual conventional computing of left to right as long as the order is irrelevant. To this end, this assumption applied to the original SyReC synthesis, which calculates the lines λ_E from Equation 5.3 as a total sum of lines. On the other hand, the number of lines may differ if λ_{E_L} and λ_{E_R} are swapped within Equation5.10.

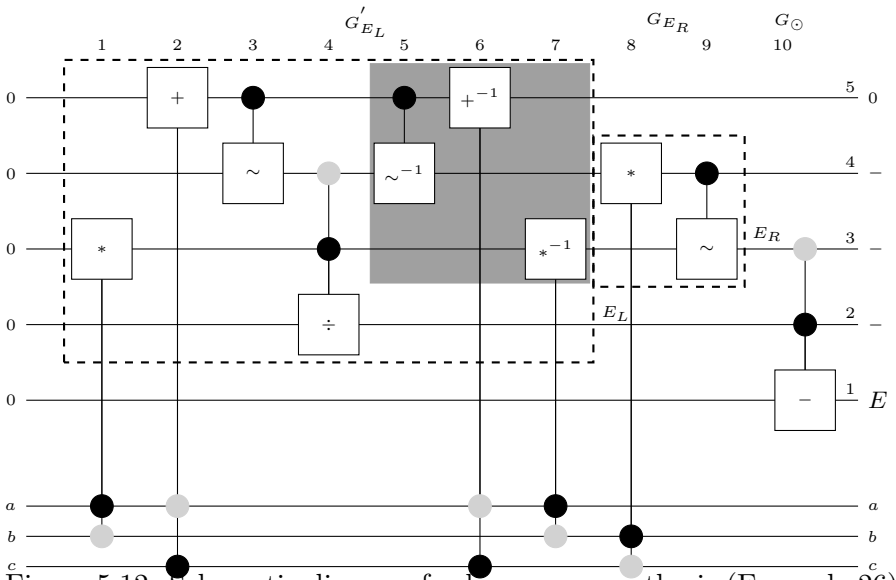


Figure 5.12: Schematic diagram for line-aware synthesis (Example 26).

Example 27. *In Example 26, the circuits are computed with* $\lambda_{E_L} = 4$ *and* $\lambda_{E_R} = 2$ *for left and right operands respectively. The total lines applied to realise the expression are* $\lambda_E = (4 + 1) = 5$*. However, if the operand circuits are swapped, then the total number of lines becomes* $\lambda_E = (4 + 2) = 6$*, according to Equation 5.15.*

It can be shown that the total constant inputs applied to realise the expression with this algorithm are lower when the operand with more lines is realised first. In this case, the second operand is computed completely with reused lines, i.e., with no extra lines (see Figure 5.13).

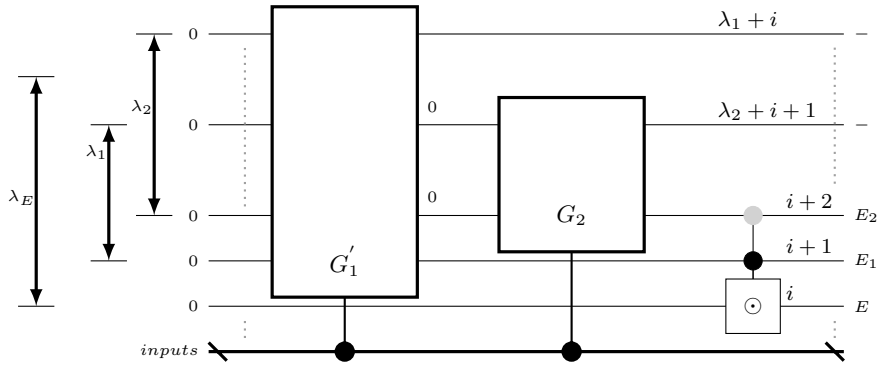


Figure 5.13: Schematic-diagram for G_E with the larger operand computed first.

Interpreting this observation for circuit realisations, with even less constant inputs, we conditionally swap the circuits G_{E_L} and G_{E_R} in a synthesis cascade, if and only if $\lambda_{E_R} > \lambda_{E_L}$. Here, Equations 5.9, 5.11, and 5.2 are modified to [1]

[1] This modification is limited to binary-expressions, while unary-expressions are still being realised according to Equations 5.2 and 5.8.

$$(\ G_1\ ,\ G_2\ ,\ \lambda_1\ ,\ \lambda_2) = \begin{cases} (\ G_{E_L}\ ,\ G_{E_R}\ ,\ \lambda_{E_L}\ ,\ \lambda_{E_R}) & \text{if } \lambda_{E_L} \geqslant \lambda_{E_R} \\ (\ G_{E_R}\ ,\ G_{E_L}\ ,\ \lambda_{E_R}\ ,\ \lambda_{E_L}) & \text{if } \lambda_{E_L} < \lambda_{E_R} \end{cases} \tag{5.12}$$

$$G_{(E_L \odot E_R)} = G_1^{'}\ \ G_2\ \ G_{\odot} \tag{5.13}$$

$$G'_{(E_L \odot E_R)} = G_1^{'}\ \ G_2\ \ G_{\odot}\ \ G_2^{-1}\ \ G_1^{'-1} \tag{5.14}$$

$$\lambda_{(E_L \odot E_R)} = \begin{cases} \lambda_1 + 1, & \text{if } \lambda_1 \neq \lambda_2 \\ \lambda_1 + 2, & \text{if } \lambda_1 = \lambda_2 \end{cases} \tag{5.15}$$

Example 28. *Applying Equations 5.13, 5.14, 5.2, and 5.8 to realise the expression in Example 26 realises the circuit with only* ***4*** *constant lines applied to its inputs, i.e., with one line less because of operands' reorder (see Figure 5.14). Here, the left operand* $E_L =$`((a * b) / ~(c + a))` *is an expression with a left operand* $E_{L_L} =$`(a * b)` *and* $E_{L_R} =$`~(c + a)` *with* $(\lambda_{E_{L_L}} = 1) \neq (\lambda_{E_{L_R}} = 2)$*. Now, applying Equation 5.15 to determine the number of lines gives* $\lambda_E = 2 + 1 = 3$ *lines applied to compute* $G^{'}_{E_L}$ *(lines* $2, 3,$ *and* 4*) as shown in Figure 5.14. The right operand of the top-level operation* E_R *requires two constant lines to be realised* λ_{E_R}*, i.e., completely computed using re-computed lines and no new line is required. The total number of applied constant inputs to compute the expression is* $\lambda_E = \lambda_1 + 1 = (3 + 1) = 4$ *as shown in Figure 5.14.*

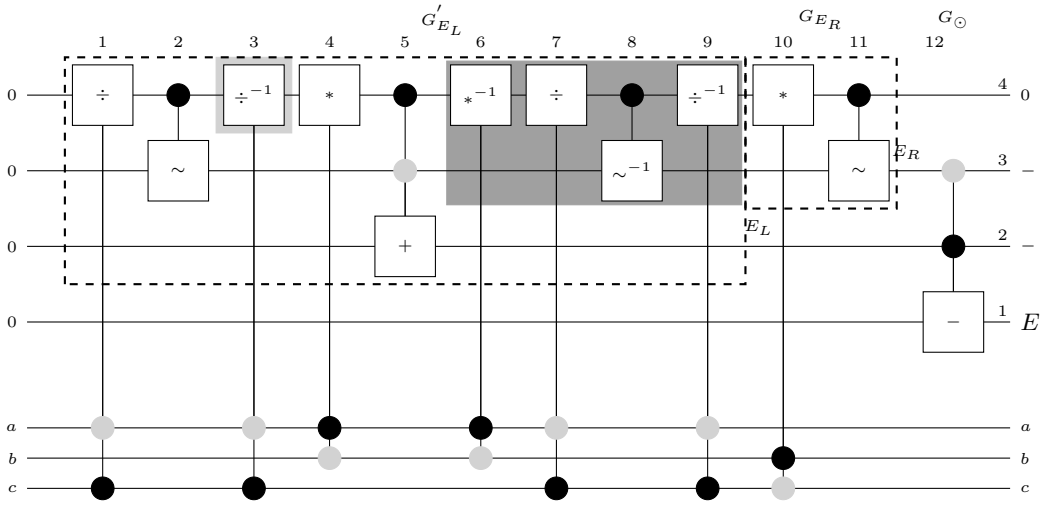


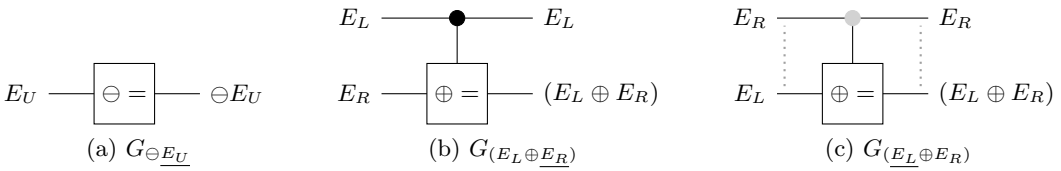
Figure 5.14: Line-aware synthesis with operands' reorder (Example 28).

It might be considered an insignificant reduction of only one line, but it provides a 20% improvement in this example (1/5 lines). The recursive nature of this procedure gives operand reorder the possibility to reduce a line for each stage in the recursion. This may result in many lines reduced from the overall realisation of the circuit depending on the shape of the expression tree.

5.2.4 Reversible Operators

SyReC realises expression operations with the assumption that they are irreversible (see Section3.2.2), which allows the synthesis of any operation by applying a constant $'0'$ line (see Figure 5.3). This assumption ignores some operations that are reversible, such as `(+,-,^,~,!)`, which are not many, but intensively used. This exceptional reversibility allows updating the operand instead of applying a constant input to compute the operation (see Figure 5.15).

On the gate level, this reversible realisation is similar, to replacing irreversible binary operators with their equivalent reversible assignments in the source code, e.g., replacing + with +=. Hence, we will use $\oplus =$ and $\ominus =$ in the remainder of this chapter for binary and unary operations, respectively, whenever they are reversibly realised.



(a) $G_{\ominus \underline{E_U}}$ (b) $G_{(E_L \oplus \underline{E_R})}$ (c) $G_{(\underline{E_L} \oplus E_R)}$

Figure 5.15: Schematic representation of reversible operators.

The updated operands in Figure 5.15 are underlined. Figures 5.15(b) and 5.15(c) are computing the same expression with the only difference being the updated operand. The updated operands should never be a signal identifier because this would violate the assumption of unchanging inputs while computing the expression. In other words, this cannot be applied if both operands are signal identifiers, e.g., `(a + b)`, where both `a` and `b` are signal identifiers. This reversible computation changes the entire realisation, consequently, as all equations describing the circuit are changed. This modification can realise expressions with fewer lines as well as with less cost.

5.2.4.1 Unary Expressions

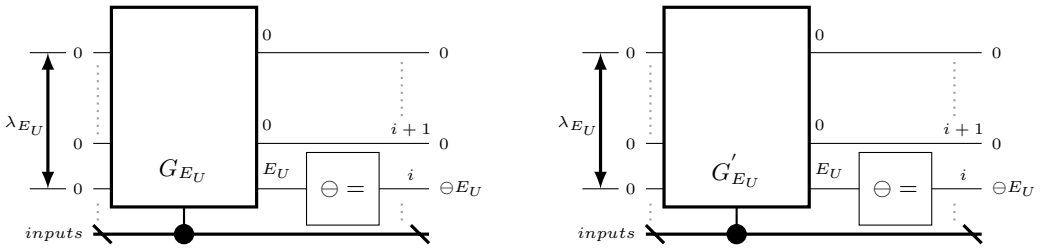
All unary expressions are reversible because the operations `~` and `!` are reversible. A reversible computation updates the line to which the operand is computed, as follows:

$$G_{\ominus E_U} = G_{E_U} \;\; G_{(\ominus =)} \tag{5.16}$$

$$G'_{\ominus E_U} = G'_{E_U} \;\; G_{(\ominus =)} \tag{5.17}$$

Equations 5.16 and 5.17 differ in that realising the operand with a garbage-free circuit, G'_{E_U}, results in a garbage-free realisation $G'_{\ominus E_U}$, and vice versa (see Figure 5.16). This means that the number of lines applied to compute the expression is used for the operand, i.e., one line less in Equation 5.18 than in Equation 5.4.

$$\lambda_{\ominus E_U} = \lambda_{E_U} \tag{5.18}$$

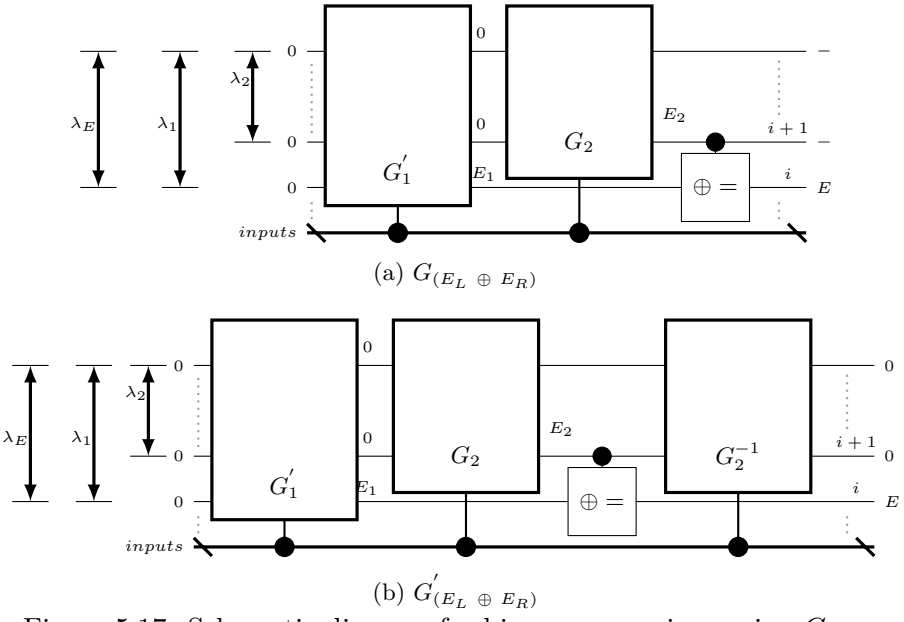


(a) $G_{\ominus E_U}$ (b) $G'_{\ominus E_U}$

Figure 5.16: Schematic diagram for unary expressions using $G_{(\ominus=)}$.

5.2.4.2 Binary Expressions

Reversible computation of $\oplus$, i.e., $G_{\oplus=}$ updates the same line by which G_1 is computed, and, hence, this operand should not be re-computed at the end of the cascade of garbage-free circuits, (see Figure 5.17). Such circuits compute $G'_{(E_L \oplus E_R)}$ with one line less and with even less cost, as compared to Section 5.2.3. These circuits are described by Equations 5.19, 5.20, and 5.21 instead of Equations 5.13, 5.14, and 5.15 in Section 5.2.3.



(a) $G_{(E_L \oplus E_R)}$

(b) $G'_{(E_L \oplus E_R)}$

Figure 5.17: Schematic diagram for binary expressions using $G_{(\oplus=)}$.

$$G_{(E_L \oplus E_R)} = G'_1 \;\; G_2 \;\; G_{(\oplus=)} \tag{5.19}$$

$$G'_{(E_L \oplus E_R)} = G'_1 \;\; G_2 \;\; G_{(\oplus=)} \;\; G_2^{-1} \tag{5.20}$$

$$\lambda_{(E_L \oplus E_R)} = \begin{cases} \lambda_1, & \text{if } \lambda_1 \neq \lambda_2 \\ \lambda_1 + 1, & \text{if } \lambda_1 = \lambda_2 \end{cases} \tag{5.21}$$

Example 29. *Figure 5.18 shows the same expression as in Example 28, where only* ***three*** *constant input lines are applied to the circuit. The circuit is computed with* ***ten*** *operations as compared to* ***twelve*** *in Example 28.*

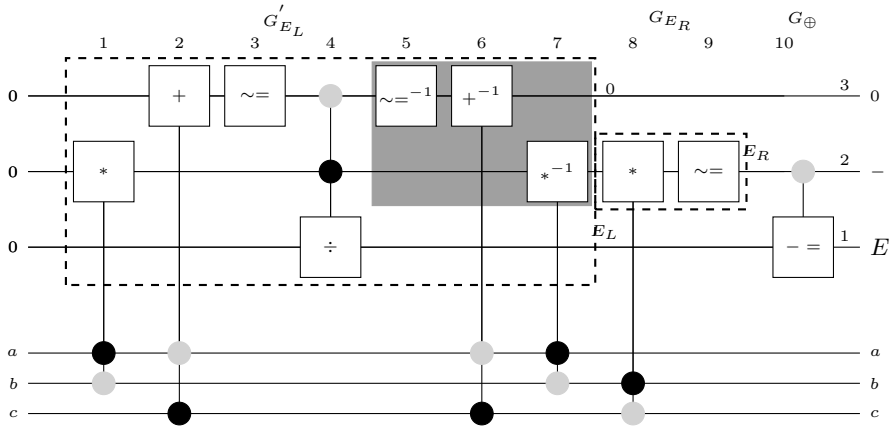


Figure 5.18: Computing the expression in Example 29.

Examples 25, 26, and 28 compute the same expression using three different, yet equivalent, realisations, which demonstrate the cost-for-lines trade-off as the circuit with fewer lines is realised with higher cost (more operations) as presented by the curve in Figure 5.20. This figure shows that Example 29 is an exception as it falls below the trade-off curve meaning that, in a reversible paradigm, reversible computations are easier than non-reversible computations.

5.3 Cost-aware Synthesis

In the reversible paradigm, circuit synthesis approaches have the two optimisation trade-off objectives of the desired function with as few circuit lines and with as low gate cost as possible. Section 5.2 introduced circuit lines as the main objective for optimization, and promotes line-aware realisations. These realisations are trading reduced lines with extra cost, except in Section 5.2.4, where a reduction in lines is not traded in with extra cost.

Cost is the other important circuit metric. First, we discuss the impact of line-aware realisations on circuit cost followed by an investigation of possible modifications to avoid or reduce cost trade-offs without compromising the reduced number of lines obtained in Section 5.2.

5.3.1 Cost-line Trade-offs

Figures 5.5, 5.12, and 5.14 show three different realisations to compute the same expression E in Example 22. The lines and cost parameters of each realisation is shown in Figure 5.20 in which we can see a trade-off curve for these realisations. The cost metric used in this graph is the overall number of operations β used to compute the expression [2], including inverse-computations (re-computation). Since all realisations are similar in the number of operations (for the same expression), the difference in the total number of computations arises from the inverse-operations. Here, we know for certain that in a garbage-free circuit, $\beta_{G'_E} \geqslant \beta_{G_E}$. In this case realising the operand with the lower cost first results in less overall circuit cost. Consequently, Equation 5.12 is modified such that the operands are rearranged to compute the one with lower cost first when both are realised with the same number of constant inputs:

$$(G_1, G_2, \lambda_1, \lambda_2) = \begin{cases} (G_{E_L}, G_{E_R}, \lambda_{E_L}, \lambda_{E_R}) & \text{if } \lambda_{E_L} > \lambda_{E_R} \\ (G_{E_R}, G_{E_L}, \lambda_{E_R}, \lambda_{E_L}) & \text{if } \lambda_{E_L} < \lambda_{E_R} \\ (G_{E_R}, G_{E_L}, \lambda_{E_R}, \lambda_{E_L}) & \text{if } (\lambda_{E_L} = \lambda_{E_R}) AND (\beta_L > \beta_R) \end{cases} \tag{5.22}$$

5.3.2 Partial (incomplete) Re-compute

Line-aware realisation of expressions, as proposed in Section 5.2, re-computes garbage lines applied to compute an operand for reusing these lines in computing the other operand in a binary expression. So all garbage lines are re-computed. The algorithm rearranges the order of computing the operands such that the operand G_1 is realised before G_2 when $\lambda_1 \geqslant \lambda_2$ (see Section 5.2.3). If $(\lambda_1 - \lambda_2) \leqslant 1$, then all re-computed lines are reused to realise G_2, but when $(\lambda_1 - \lambda_2) > 1$, there exist re-computed line(s) that are not reused, e.g., in Figure 5.18 where line 3 is re-computed but not reused. In this case, inverse operations add cost for no reason. This superfluous cost can be identified and reduced, by removing these inverse-operations from the circuit, as follows:

1. If the expression is binary ($EL_{\odot}E_R$), and $(\lambda_1 - \lambda_2) < 1 \leqslant l$, then the circuit incorporates $(l-1)$ lines that do not have to be re-computed.

2. To identify an operation that is unnecessarily re-computing a line:

   (a) The operation is a part of a re-compute circuit G_x^{-1} of the first operand G'_1 (the space with grey background).

   (b) The line to which this operation is computed is not used after this operation, neither as a control nor as a target line.

[2] Operation count is used as a cost indicator assuming each operation has the same unity cost metric. This assumption is very far from being accurate as there are considerable differences in the complexity of operators' circuits. This assumption provides a reasonable operator-independent assessment to algorithms efficiency in this context.

3. To remove these inverse operations, start from the last operation within G_x^{-1}:

   (a) Check if it is removable,

   (b) if yes, remove and go to the operation before, and

   (c) repeat until the first operation in the re-compute circuit is checked.

Example 30. *In Figure 5.18, the left operand G'_{E_L} is realised by applying $\lambda_L = 3$ constant, and the right-operand G_{E_R} is realised by applying $\lambda_R = 1$, $(\lambda_L - \lambda_R) = 2$, i.e., there exists $(2 - 1 = 1)$ unnecessarily re-computed line (line 3). The shaded area in this figure shows the inverse computation of garbage lines of the left operand, i.e., operations $(5, 6,$ and $7)$. Figure 5.19 is identical to Figure 5.18. Starting from 7 back to 5, the operation is checked. Here, operations 6 and 5 are both re-computing line 3, which is not used in the rest of the circuit, so both operations are removed.*

This incomplete operand re-compute modifies the proposed realisation to improve circuit cost without compromising the minimum line achieved by line-aware realisation, as proposed in Section 5.2 (see Figure 5.20).

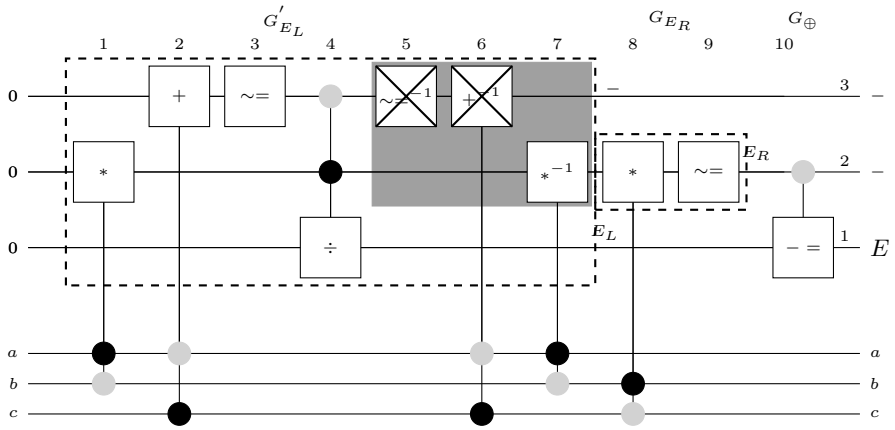


Figure 5.19: Block diagram of the expression in Example 22 with garbage partially re-computed.

5.4 Manipulating Expressions

Sections 5.2 and 5.3 propose different ways to compute a SyReC expression and show the impact of each algorithm on the number of lines and costs of the resulting circuits. One observation is that the order of computing the operations does matter (see Section 5.2.3). This motivates investigating the impact of the expression-tree shape on the resulting circuits.

In this section, we discuss further improvements on circuits by manipulating the order of operations in computing expressions by considering properties of operations, such as the associative and commutative properties.

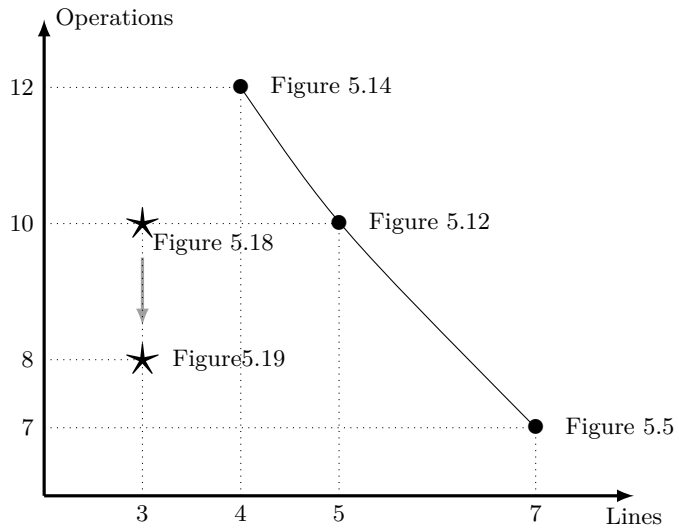


Figure 5.20: Lines vs. cost metrics of circuits in Figures5.5, 5.12, 5.14, 5.18, and 5.19.

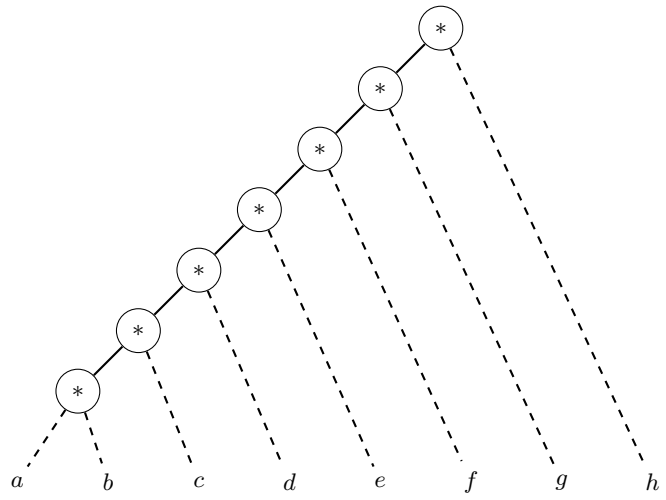
Example 31. *Figure 5.21 shows trees of three different expressions, which compute the product of eight signals. The expressions are mathematically equivalent to each other because multiplication (⋆) is associative:*

1. *Figure 5.21(a) shows* E_1 = `((((((a⋆b)⋆c)⋆d)⋆e)⋆f)⋆g)⋆h)` *in which each operation has one operand as a primary signal.*
2. *Figure 5.21(b) shows* E_2 = `((a⋆b)⋆(c⋆d))⋆((e⋆f)⋆(g⋆h)))` *in which each operation has two operands with the same size(balanced tree).*
3. *Figure 5.21(c) shows* E_3 = `(((a⋆b)⋆c)⋆(d⋆e))⋆((f⋆g)⋆h))` *in which each operation has the right operand realised using exactly one line less than the left operand.*

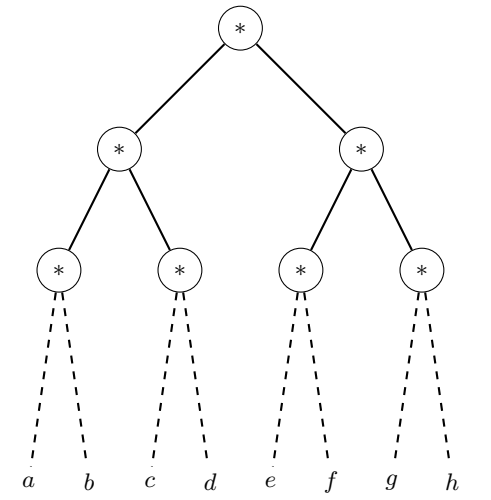
Despite being equivalent, these expressions are realised with different circuits (see Figure 5.22):

1. *Figure 5.22(a) shows* E_1 *computed using 7 lines and 7 operations. The realisation is considered as the worst case in terms of constant inputs, where no operation is re-computed.*
2. *Figure 5.22(b) shows* E_2 *computed using 5 lines and 9 operations. This balanced structure shows a moderate reduction in lines and trade-off with some increase in cost.*
3. *Figure 5.22(b) shows*E_3 *computed using 4 lines and 12 operations, which is considered as the best case with respect to the number of lines and with extra cost.*

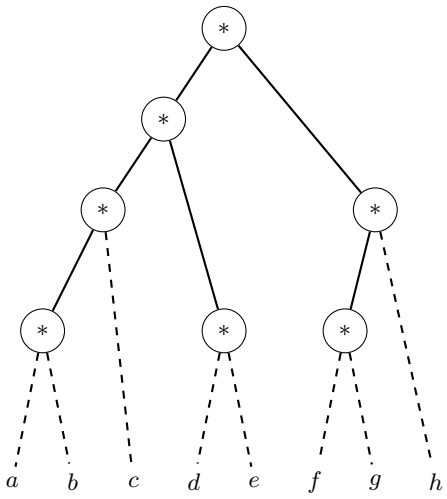
Replacing multiplication (⋆) with addition (+) results in other expressions (E_1'*,* E_2'*, and* E_3'*), which compute the summation of the signals rather than the product. These expressions have similar tree structures to their counterparts (i.e.,* E_1*,* E_2*, and* E_3 *respectively), yet the computations are obviously different. The fact that (+) can be replaced by the reversible (+=) changes the realisations, as shown in Figure 5.23.*



(a) $E_1 = (((((((a * b) * c) * d) * e) * f) * g) * h)$



(b) $E_2 = (((a * b) * (c * d)) * ((e * f) * (g * h)))$



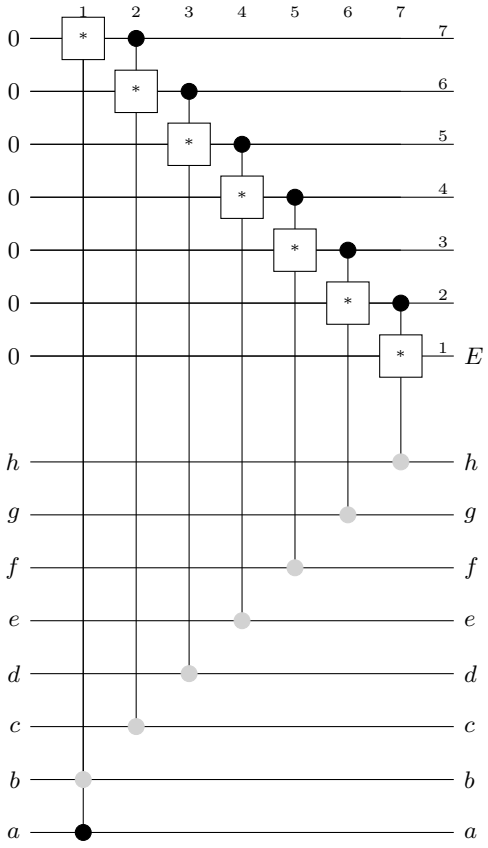
(c) $E_3 = ((((a * b) * c) * (d * e)) * ((f * g) * h))$

Figure 5.21: Three different equivalent expressions.

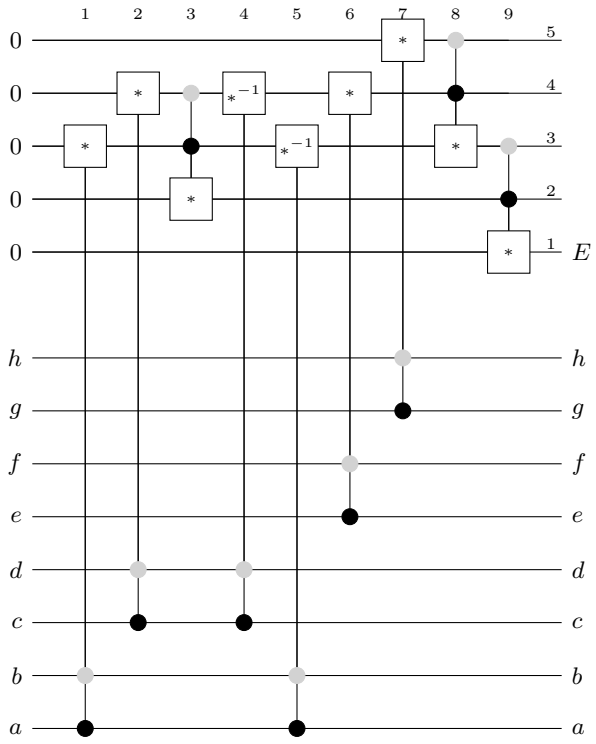
1. *Figure 5.23(a) shows* $E_1' =$ `(((((((a+b)+c)+d)+e)+f)+g)+h)` *computed in the best possible way with reversible operations, and only one constant input is exploited in this realisation with the summation using 1 constant line and 7 operations. On the other hand, the expression* E_1*, which has the same tree structure is considered the worst case in terms of lines.*

2. *Figure 5.23(b) shows* $E_2' =$ `(((a+b)+(c+d))+((e+f)+(g+h)))` *with a perfectly-balanced tree structure with the worst realisation, in terms of lines. The summation exploits 3 lines and 8 operations.*

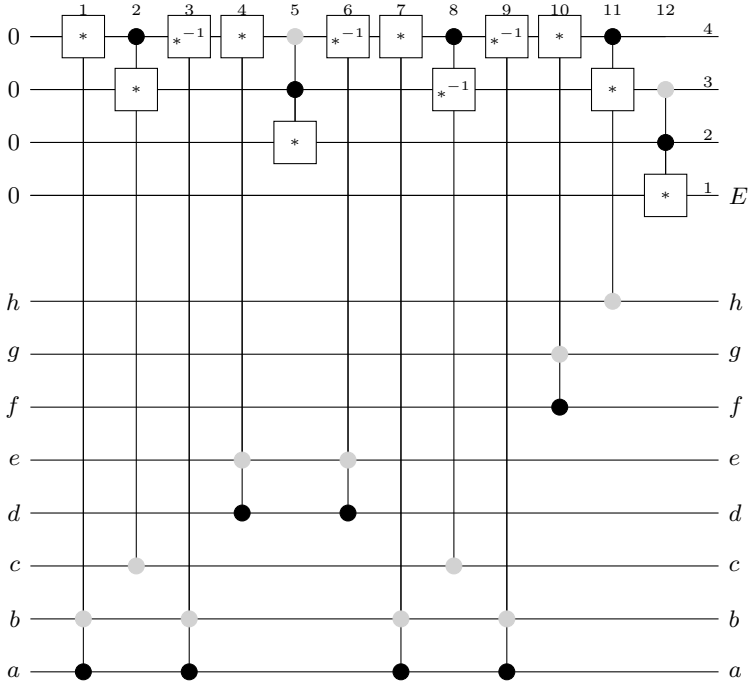
3. *Figure 5.23(c) shows* $E_3' =$ `(((a+b)+c)+(d+e))+((f+g)+h))` *computed using 2 lines and 8 operations, while the same tree structure resulted in the best case to compute the product (i.e., in* E_3*).*



(a) Worst case expression E_1

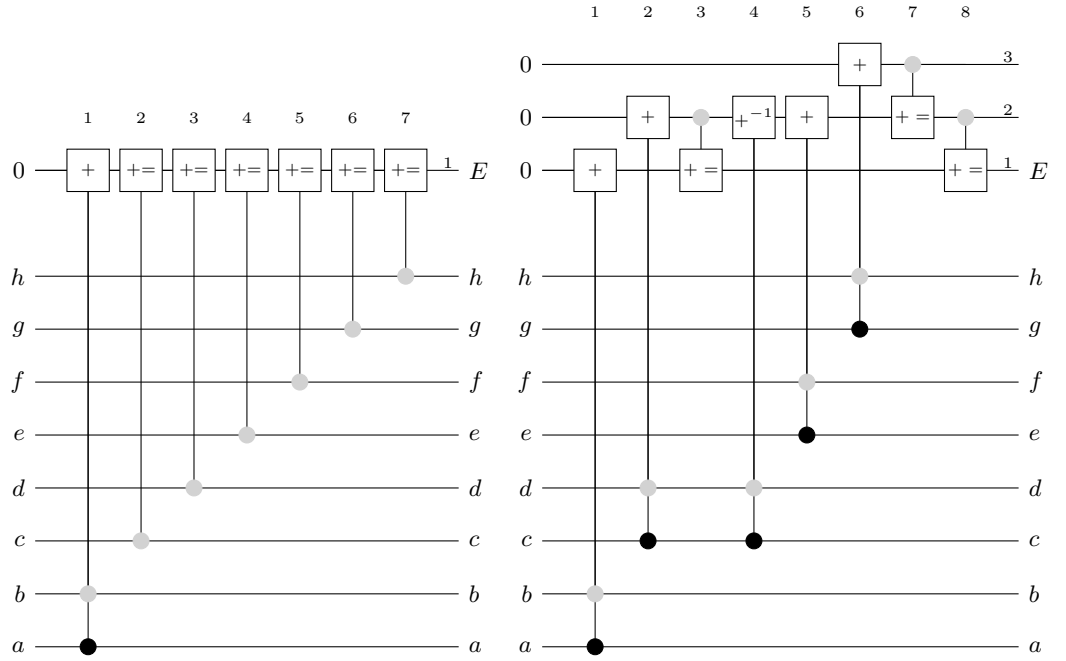


(b) Perfectly-balanced expression E_2

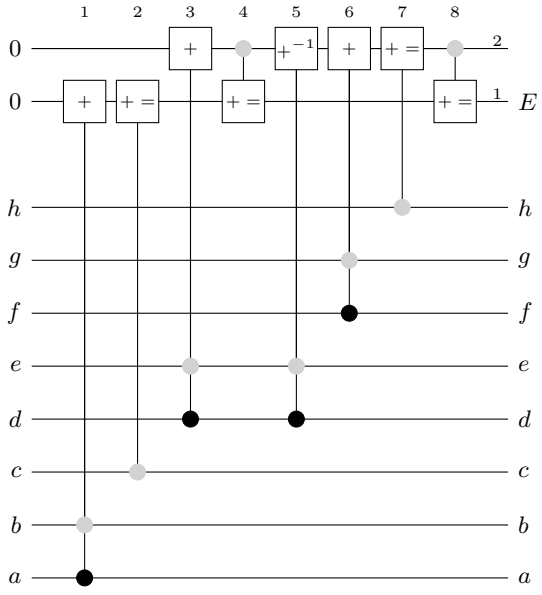


(c) Best case expression E_3

Figure 5.22: Computing the expressions of Figure 5.21.



(a) $E_1' = (((((((a+b)+c)+d)+e)+f)+g)+h)$ (b) $E_2' = (((a+b)+(c+d))+((e+f)+(g+h)))$



(c) $E_3' = ((((a+b)+c)+(d+e))+((f+g)+h))$

Figure 5.23: Computing trees with reversible operations.

To generalise, manipulating expressions, whenever possible, can result in better circuits, for which we can observe the following:

1. Figure 5.21(a) shows an expression arranged such that, for each operator, at least one operand is a primary signal. This case is identical to the original SyReC algorithm

as no re-compute occurs. Here, $(\lambda = k = n = b)$, where k is the number of operators in an expression, λ is the number of constant $'0'$ circuit lines, b is the number of operations and n is the height of the binary tree. This is the worst case with respect to the number of lines, as the results show. However, when such a tree structure is to compute reversible operations, such as (+=), the entire expression may be computed using only 1 line, which is the best possible case.

2. Figure 5.21(b) shows a perfect binary tree, which exploits lines to be synthesized. Here,

$$\lambda = 2(\lceil log_2(k+1) \rceil) - 1 = 2 \cdot n - 1 \tag{5.23}$$

where λ is logarithmically instead of linearly dependent on k. Practically, a perfect binary tree is a special case that can occur only when the expression is composed using $(k = 2^n - 1)$ binary operators. Again, n is the height of the tree. The cost is expected to increase exponentially as a trade-off where, in this case, the number of operations b is given by

$$b = 3^{(\lceil log_2(k+1) \rceil - 1)} = 3^{n-1} \tag{5.24}$$

The same tree structure results in circuits that provide no advantage when the operations are reversible (e.g., with +).

3. Figure 5.21(c) shows a best-case tree where the highest efficiency is obtained from line-aware synthesis in terms of the number of lines. For all operations, the right-hand operand requires exactly one circuit line less than the left operand. Hence, right operands completely reuse lines and do not need to allocate another line. This case shows the maximum usage of re-computed lines. Here, $\lambda = n$ and the number of operators, k, in this case is recursively calculated using the sequence [3]

$$k = \kappa(n) = \kappa(n-1) + \kappa(n-2) + 1 \tag{5.25}$$

where $\kappa(1) = 1$ and $\kappa(2) = 2$. For example, if there are $\lambda = 6$ lines, then $\kappa(6) = 20$, i.e., an expression arranged in the best case with up to $k = 20$ operations can be calculated by using these lines. The total number of operations to compute the expressions is determined from the sequence

$$b = \beta(n) = 2 \cdot \beta(n-1) + \beta(n-2) \tag{5.26}$$

Manipulation is not always possible, but when it is, it would be better to have the maximum benefit from the line-aware expression synthesis algorithms.

[3] This sequence is strongly related to the *Fibonacci sequence* $f(n) = f(n-1) + f(n-1)$ where $\kappa(n) = f(n+2) - 1$

5.5 Experimental Evaluation

Expression realisations, as proposed in this chapter, are experimentally evaluated and discussed in this section. The aim is to assess the impact of the different ways to compute an expression, in practice, on the resulting circuits.

The discussion in this chapter is carried out on a higher-level of abstraction, which is comparable to the register-transfer level in conventional design flow, instead of at the gate-level. We deal with variable length unsigned integers as the data unit for which the operations are defined. The usual gate-level metrics, as discussed in Section 2.2.5, become confusing and not useful in evaluating the quality of the computations because they are related to gate-level details, such as operation bit-widths and the gate complexity of different operations' circuits. Therefore, we suggest two metrics, which are coherent with the level abstraction in this chapter:

1. Constant inputs (λ): The number of constant $'0'$ lines applied to the circuit to compute the expression, Signal input lines are excluded from the count because they are not subject to optimisation. Then, the number of lines are divided by the operator bit-width in order to have a bit-width-independent line metric.

2. Operation blocks (β): The number of operations and inverse operations performed to compute the expression. Each operation is identified as a unit circuit (a basic block) regardless of the actual gate cost of this specific operator circuit by which we can have an operation independent measure for the cost expansion.

The experiment carried out using a set of benchmark expressions with different sizes and structures. Each expression is realised five times using the following configurations:

(R1) Direct synthesis as originally performed by SyReC synthesiser, as explained in Section 5.1.2.

(R2) A line-aware synthesis as proposed in Section 5.2.2.

(R3) A line-aware synthesis with rearranged operands, such that the first is computed first, as proposed in Section 5.2.3.

(R4) Scenario (R3) modified to incorporate reversible updates when computing special operations, i.e., $\oplus=$ and $\ominus=$, as proposed in Section 5.2.4.

(R5) Scenario (R4) with incomplete re-compute of garbage lines, as proposed in Section 5.3.2.

Table 5.1 shows λ and β measured for each realisation as well as the number of operators used to compute the benchmark k. From this table, we observe:

1. all configurations have the same results for small expressions (three or fewer operations). On the other hand, large expressions show significant differences between circuits in both λ and β configurations. It is true that complex expressions are not the trend in

HDL programming, but such expressions may appear, e.g., as complex conditionals. Having just one such expression in the entire module is enough to set a lower bound to the constant inputs applied to the circuit.

2. (R5) is always better than (R3) and (R4) in both measures. Consequently, the only three alternatives proposed are (R2), (R5), and the original (R1). Therefore, the comparison graphs in Figure 5.24 and Figure 5.25 include only the realisations R1, R2, and R5.

3. (R1) always has the lowest β. Consequently, there is no need to use another alternative except for the largest expression in a module, i.e., the most line demanding expression, which sets a bound for the number of constant inputs in the entire circuit. Here, it is better to compute the expression with fewer lines.

4. (R5) always has the lowest λ. Therefore, it is the one to be used for the largest expression in a module. On the other hand, configuration (R2) may have some advantage over (R5) in cases when it computes with less β if λ is good enough.

5. The reduction of constant lines in configurations (R2) and (R3) depends on the number of lines in the two operands' sub-circuits. Maximum reduction is obtained when the right operand is realised using one line less than the left operand ($\lambda_{E_L} = \lambda_{E_R} + 1$).

6. Scenario (R4) shows a reduction in both parameters without any trade-off only when the expression contains some reversible operators (`!` , `~` , `^` , `-` , `+`); otherwise, it has no impact on the circuit.

7. Scenario (R5) shows a tangible reduction in cost as compared to scenario (R3) when there is a large difference in the number of lines between the two operands' sub-circuits ($\lambda_{E_L} \gg \lambda_{E_R}$) or ($\lambda_{E_L} \ll \lambda_{E_R}$).

8. From observations 5 and 7 above, we can better understand the influence of the shape of an expression tree on the circuits, which was discussed in Section 5.4. Taking this into consideration, programmers may improve writing their expressions such that:

   (a) binary expressions with the form $((E_1 \oplus E_2) \oplus S)$ instead of $(E_1 \oplus (E_2 \oplus S))$, where E_1, E_2 are two expressions and S is a primary signal. This suggestion maps to better circuit costs and fewer lines in some cases.

   (b) expressions arranged in the worst case are avoided. It is much better to have an expression written as $(E_1 \odot (E_2 \odot S))$ instead of $((E_1 \odot E_2) \odot S)$, where E_1 is a large expression, E_2 is smaller and S is a primary signal. It is better, whenever possible, to split the expression such that one operand is slightly smaller than the other. More precisely, it is better to have a smaller operand sub-expression with only one line less than the larger operand sub-expression.

Table 5.1: Experimental evaluation for different benchmark expressions' circuits.

Benchmark		(R1) Section 5.1.2		(R2) Section 5.2.2		(R3) Section 5.2.3		(R4) Section 5.2.4		(R5) Section 5.3.2	
Name	k	λ	β	λ	β	λ	β	λ	β	λ	β
ov4	1	1	1	1	1	1	1	1	1	1	1
acc	2	2	2	2	2	2	2	2	2	2	2
avsp	3	3	3	3	3	3	3	3	3	3	3
p2o	4	4	4	3	5	3	5	3	5	3	5
maj3	5	5	5	4	7	4	7	4	7	4	6
p3o	5	5	5	4	6	4	6	2	6	2	6
f3o	6	6	6	5	7	4	8	4	8	4	8
exp7	7	7	7	5	10	5	10	3	8	3	8
exmp7	7	7	7	5	10	4	12	3	10	3	18
f4o	8	8	8	6	15	5	17	5	17	5	15
exp9	9	9	9	7	17	6	21	6	21	6	16
av10	10	10	10	7	14	6	18	4	12	4	12
exp11	11	11	11	7	17	6	23	6	23	6	18
f8o	16	16	16	9	129	8	145	8	145	8	66
se7	17	17	17	8	30	7	42	4	28	4	27
maj5	29	29	29	9	114	9	144	9	114	9	111
p8o	44	44	44	10	135	9	167	6	94	6	92

(k) is the number of operations in the benchmark expression.
(λ) is the number of constant lines exploited to compute the expression.
(β) is the total number of operations and inverse-operations (re-compute) blocks used to compute the expression.

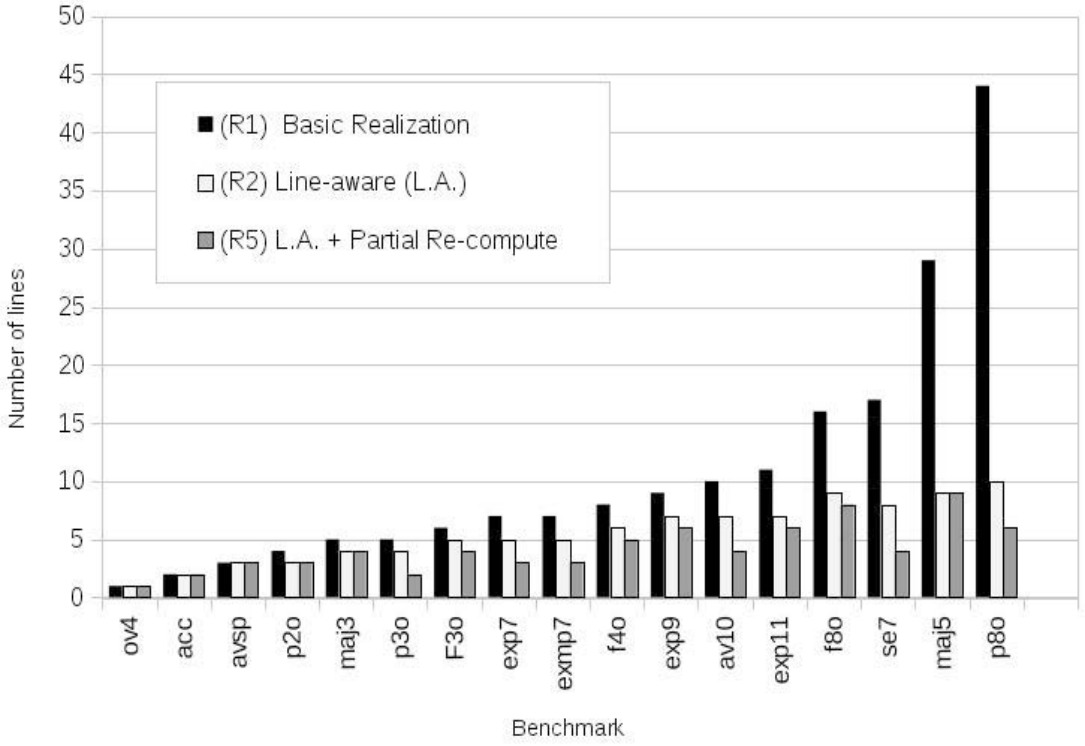


Figure 5.24: Constant lines exploited to compute expressions in different realisations.

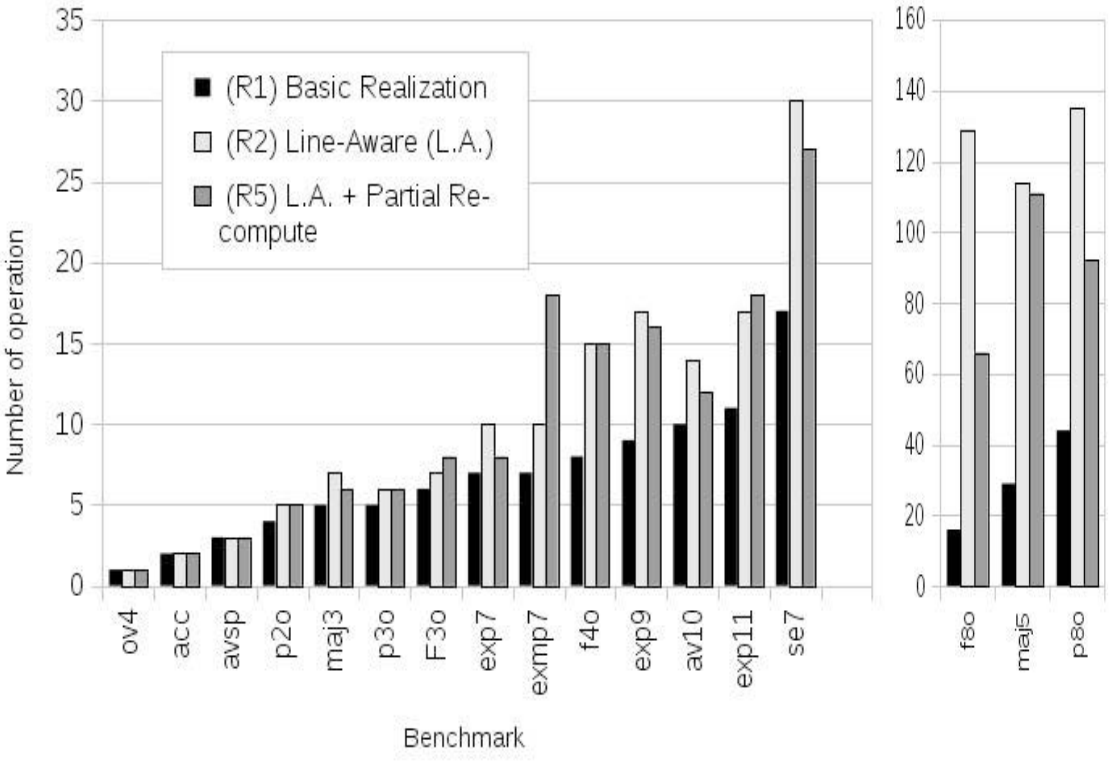


Figure 5.25: Costs of computing expressions in different realisations.

5.6 Summary

In this chapter, various procedures are proposed to realise SyReC expressions. The main objective is to compute expressions with as few constant inputs as possible, without the need to change the programming style or compromise simplicity of HDL descriptions.

The secondary optimisation objective is to avoid the expected cost-for-line trade-off, if possible. This line-awareness is based on reusing lines to avoid applying new ones. The experimental results show that in many cases the proposed algorithms succeeded in reducing the number of lines with relatively small increases in circuit cost.

Different realisations are discussed, with three proposed as candidates for use as alternatives to compute expressions. The basic realisation as originally defined in SyReC (i.e., without operation re-compute) is discussed in Section 5.1.2, and this realisation shows the minimal cost. Hence, this approach is recommended when the circuit already provides a sufficient number of constant inputs.

The realisation proposed in Section 5.3.2 is the one to compute expressions with fewer lines in most cases. Consequently, it is recommended when circuit lines are limited (e.g., when computing the largest expression in the module). In a few cases, the realization proposed in Section 5.2.2 shows better results as compared to Section 5.3.2. Therefore, this approach may also be considered as an alternative.

Realising reversible circuits that compute HDL-expressions as proposed in this chapter is not exclusively applicable for SyReC. In fact, SyReC considers irreversible operations in its expressions. Hence, the same approaches may be used to compute conventional HDL-expressions if we can define reversible circuits for the operations of that HDL.

Chapter 6

Synthesis of Reversible Circuits Using Conventional HDL

An elaborated flow emerged over the last few decades for the design of circuits and systems based on the conventional computation paradigm. This flow is composed of several levels of abstraction (e.g., specification level, electronic system level, register transfer level, gate level, and transistor level). A wide range of design tools has been developed and accepted as standards in the industry, such as modelling languages, system description languages, and hardware description languages [18]. Furthermore, these design methodologies are additionally supported by various powerful approaches for simulation, verification, validation, and debugging to ensure the correctness of a designed circuit or system. On the other hand, considering the reversible computation paradigm, none such powerful approach is available so far. Although researchers have considered the basic tasks of synthesis, verification, and debugging, reversible computation design flow is still elementary.

Many efforts have been made to investigate the suitability of conventional computation methodologies for circuit design in the reversible computational paradigm at the gate level description, such as Boolean expressions, truth tables, and binary decision diagrams [3,4,52]. However, limited efforts are tackling the problem of reversible circuit design at a higher abstraction level. In comparison to matured conventional HDLs, SyReC is a preliminary language with one data type and a basic set of operations. In other words, the existing design methods for reversible circuits and systems are far from modern standards and industrial needs. Consequently, investigating conventional HDL approaches for the design of reversible circuits may be advantageous over dedicated reversible HDL. However, very little exists in the literature referring to conventional HDLs as a basis for reversible circuit design [63].

There are challenges discouraging researchers from considering conventional HDLs for reversible circuit synthesis, including differences between the two computational paradigms and special characteristics and restrictions applied to reversible circuits, such as:

- fanout and feedback are not directly allowed in reversible circuits [70]. Special arrangements should be made to detect and handle through feedback when these occur in the code.
- the elementary gate library used in a conventional circuit is entirely different from those used in the reversible paradigm, such that all elementary defined conventional circuits should be redefined using reversible gates.
- most of the practically relevant operations are irreversible and need some arrangement to be computed in the reversible paradigm. This is usually solved by applying constant inputs, like SyReC or other hierarchical reversible synthesis approaches.
- the concurrent nature of signal processing in conventional hardware versus the sequential nature of signal updates following cascaded gate structure in the reversible paradigm.

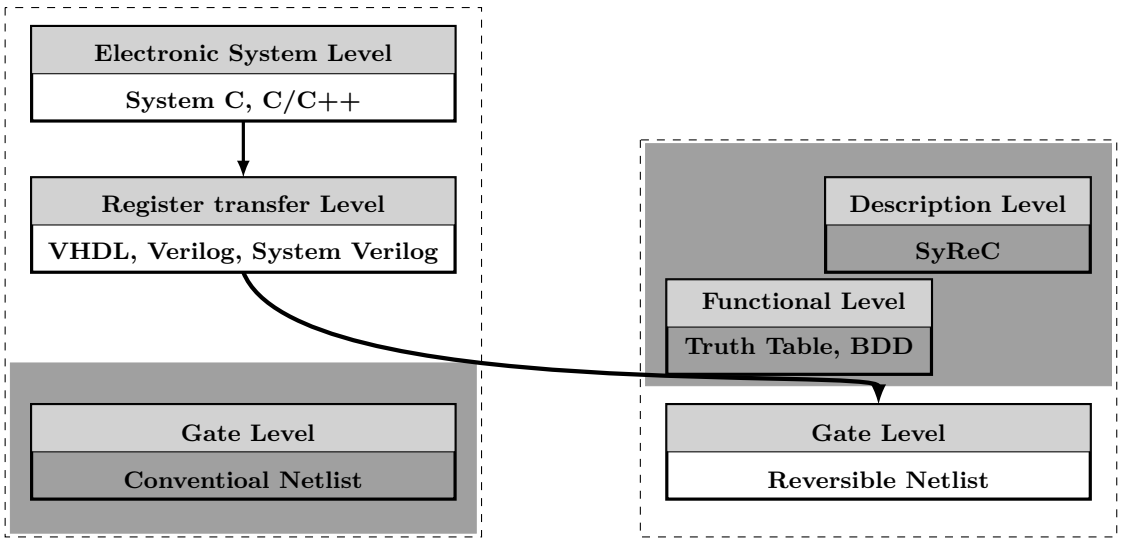


Figure 6.1: A hybrid flow that incorporates conventional tools for reversible circuits.

In this chapter, we address conventional HDL-based reversible circuit synthesis. The widely-used hardware description language, VHDL, and its suitability to synthesise reversible circuits are investigated. The findings provide the basis towards a design flow that requires no or little knowledge of the reversible computation paradigm (see Figure 6.1). At the same time, it pinpoints to the weaknesses and open issues to be addressed to make VHDL-based design an accessible alternative to the existing design solutions for reversible circuits.

6.1 Introducing VHDL

VHDL is a hardware description language designed to allow for the description of the structure of a circuit, i.e., its decomposition into subsystems as well as the interconnections, by utilising an established programming style and different levels of abstractions. Using VHDL, circuits can be simulated, synthesised, and verified before being manufactured [18]. More precisely, a circuit is first defined by an *entity* declaration, which introduces a name for the entity and lists the ports (input and output signals). An entity declaration describes the external view of the design.

The internal implementation of an entity is provided in an *architecture* body of that entity. Architectures might be provided in different fashions. A *behavioural* architecture body describes the function in an abstract way, e.g., in terms of *process statements*. A process statement defines a sequence of operations to be executed when the circuit is simulated. A wide variety of actions might be included within a process statement, which, in some cases, restricts the synthesis of the architecture.

Synthesis-oriented programmers prefer an alternative model to describe the implementation of an entity, which is called *structural* description. This model describes the circuit in terms of a net-list of sub-circuits. More precisely, sub-circuits are declared as *components*. Multiple *component instances* (i.e., copies) may appear in the architecture body to represent these subsystems. A component instance includes a *port map* to specify the interconnections of these component instances within the enclosed architecture body.

```
1          entity main is
2            port( q,r,s: in bit; y: out bit);
3          end entity main;
4
5          architecture structural of main is
6            component sub is
7              port(a,b: in bit; f: out bit);
8            end component sub;
9            signal t: bit;
10         begin
11 L1:       test port map (a => q, b => r, f => t)
12 L2:       test port map (a => t, b => s, f => y);
13         end architecture structural;
```

Figure 6.2: Structural VHDL architecture.

Another possible description used *signal assignment* statements, which define the flow of data to compute signals. An architecture body described completely using signal assignment statements is typically referred to as a *data-flow* description style. Often it is useful to describe the required system using a mixture of processes, interconnected components, and signal assignment statements.

In the remainder of this chapter, we focus on the main descriptions provided by VHDL to utilise reversible circuit synthesis. Simulation related descriptions, including process statement actions, are not covered. Because feedback connections are not allowed in the reversible circuit paradigm, circuits that contain feedback are also not supported.

```
1          entity sub is
2            port (a,b: in bit; f: out bit);
3          end entity test;
4
5          architecture data_flow of test is
6            signal x: bit;
7          begin
8  S1:       x <= not b;
9  S2:       f <= a and x;
10         end architecture data_flow;
```

Figure 6.3: Data-flow in VHDL architecture.

Example 32. *Figure 6.3 provides an example of a data-flow VHDL circuit. The entity named* `sub` *has three single-bit ports, namely two input ports (*`a,b`*) and one output port (*`f`*). A single-bit wire signal (*`x`*) is declared within the architecture body. The implementation of this system contains two signal assignment statements. The first statement (S1) computes the wire signal* `x`*, while the second statement (S2) computes the output signal* `f`*. Figure 6.2, on the other hand, shows a structural description of a VHDL architecture (*`main`*), in which the entity* `sub` *defined in Figure 6.3 is declared and instantiated twice (statements* `L1` *and* `L2`*). The* `port map` *associated with each instance defines the inter-connectivity of this specific component-instance within the* `main` *circuit.*

6.2 VHDL Signals in Reversible Circuits

A VHDL signal is meant as a mathematical representation of a circuit node in conventional hardware where the changing value of the signal (waveform) can be measured at any time. Circuit components should be properly interconnected to compute the desired signals to drive these nodes. This representation is no longer valid.

6.2.1 Circuit-lines of VHDL Signals

VHDL signal types can be mapped directly to signals (lines) of the reversible circuits. In Figure 6.3 we can see examples of different types of signals in a VHDL code, which are mapped to lines with different specifications as follows:

1. **Input ports** `a,b`: These lines carry input values to the circuit and remain unchanged within a circuit because such signals are only carrying information to the circuit from an external source of information.

2. **Output port** `f` : With a constant $'0'$ input, an expression is assigned to this signal (line) by a statement within the architecture body.

3. **Internal signal** `x` : This line represents an internal wire. It is similar to output ports in that it is initially constant $'0'$ and assigned in the same way. The difference between outputs and wires is that wires facilitate computing other signal(s), and then thet are considered garbage outputs.

4. **Implicit lines**: These lines are similar to internal signals in that they have constant $'0'$ inputs and constitute garbage outputs, but are not explicitly declared within the code. Such lines are mandatory to compute non-reversible operations, e.g., to compute the expression (`a and x`) in Figure 6.3, Statement `S2`.

6.2.2 VHDL Signals versus SyReC Signals

Despite many similarities, signal types in VHDL differ from SyReC because each language is oriented to describe a different circuit paradigm, including:

1. VHDL defines `inout` ports when a certain signal functions sometimes as an `in` and otherwise as an `out`, e.g., bidirectional data-bus signals. This has no relevance to the concept of `inout` signals in SyReC where `inout` means simply a signal that is valid at both ends of a circuit line. VHDL `inout` is not discussed in this context because it is related to state-dependent (sequential) digital systems, which contain feedback behaviour that is not allowed, by definition, in the reversible paradigm.

2. Multiple assignments to a signal cause a conflict in a conventional paradigm because it merely means connecting the outputs of different gates. Consequently, any signal is assigned, at the most, just once within the architecture body.

3. Since `in` signals, in VHDL, are not updated within the architecture body, and remain unchanged at the output end, which is unlike `in`-type signals in SyReC, which are considered garbage because they can be updated, or changed, within the module.

4. Signal assignments applied only to signals form non-`in` types, which are known to be constant $'0'$. In other words, assignment of an expression to a signal does not cause any information loss.

Up to this point, the reversible circuit is composed of empty lines only without any gates. In other words, for a circuit with an identity function, realising a VHDL code does nothing. Gates are added to process signals on circuit lines to compute the desired outputs as described by the statements in the architecture body.

6.3 Flow of Data with Signal-Assignment

An assignment statement, in its simplest form (`S <= E;`), is composed of three parts, a target signal `S`, an assignment operator <= and a right-hand side expression `E`. To realise a statement, two steps are to be followed:

1. Compose a sub-circuit G_E realising the expression `E` (see Section 6.3.1).

2. Use Toffoli gates to assign `E` to the target-signal to realise the overall statement (see Section 6.3.2).

6.3.1 Expressions

VHDL provides a set of operations, such as Boolean, arithmetic, and comparison. The operations are applied on operand signals to compose VHDL expressions. As much as it is considered in SyReC, most operations are irreversible. Hence, an additional line with constant inputs is applied to make an irreversible function reversible [36] leading to the implicit lines (discussed in Section 6.2). This is exactly how the reversible HDL SyReC tackles the problem [24]. In fact, most VHDL-defined operators can use circuit definitions provided for equivalent operations in SyReC. Despite differences in the grammar, defined operations, and their precedence between the two languages, which imply some modifications, the ideas proposed to realise SyReC expressions are applicable, as well, to realise VHDL expressions (see Chapter 5). Again, realising an expression `E` which is combined with N operators will implicitly add N constant lines to the circuit.

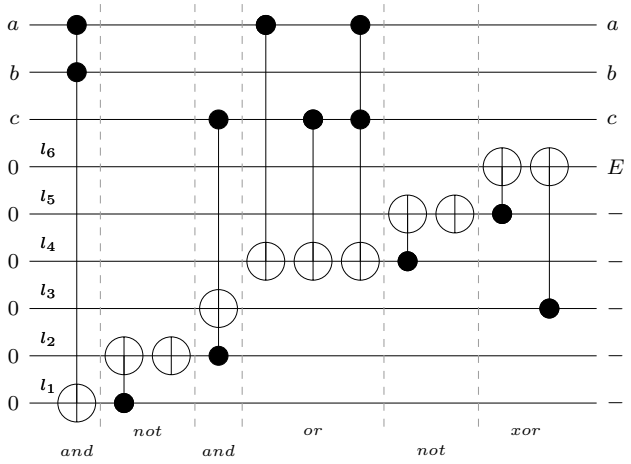
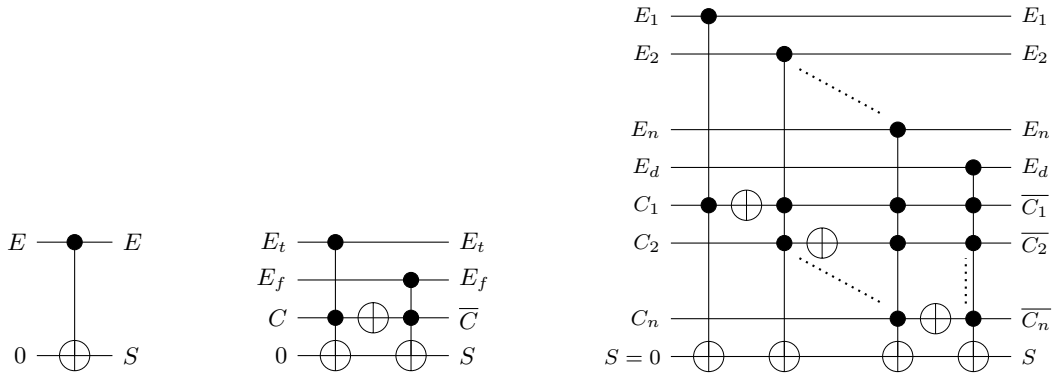


Figure 6.4: Circuit realising expression E from Example 33.

Example 33. *Figure 6.4 shows a reversible circuit to compute the VHDL expression* `E`*, which is given by (*`not(a and b) and c xor not(a or c)`*). The expression is computed based on six Boolean operations. So, six constant input lines are applied to the circuit.*

6.3.2 Assignment Operation

In contrast to SyReC, which has more than one operator to update signals reversibly, VHDL has only the signal-assignment operator, <=, that copies the line on the RHS to the LHS. A Toffoli gate can be used to copy the value of a line `E` into line `S`, if and only if, `S` is a constant $'0'$, as shown in Figure 6.5(a), because (`(E` **`xor`** `0)=E`). The operation is a simple assignment (`S <= E`) by which the expression `E` as computed in Section 6.3.1, is assigned to the target (non-input) signal `S`, which is known to be a constant $'0'$ (see Section 6.2).



(a) Simple (b) Conditional (c) Multiple

Figure 6.5: Realisation of signal assignment.

Conditional signal assignments are also provided in VHDL, with the form (`S <= Et` **`when`** `C` **`else`** `Ef`), by which `Et` is assigned to the target signal `S` only when the condition `C` is evaluated to 'true' or $'1'$, while `Ef` is assigned otherwise, as shown in Figure 6.5(b). A conditional assignment may be extended to multiple-conditionals (see Figure 6.15). A generalised arrangement of gate allows the realisation of such multiple conditional assignments, as shown in Figure 6.5(c).

6.4 Interconnecting Statements

An overall circuit realisation for a given VHDL code is computed by interconnecting sub-circuits together within one main circuit. This includes components' instances in addition to the signal assignment statements.

6.4.1 Statement Cascade

A key difference between conventional and reversible circuit paradigms is addressed here. In conventional circuits, it does not matter which statement is realised first, as the result will be the same hardware because of statement concurrency. A reversible computation paradigm, on the other hand, successively processes signals by cascaded gates. Consequently, signals are

successively (not concurrently) computed. In this regard, the order in which the statements are considered has an effect.

Signals should be prioritised according to their dependence. In other words, a signal is computed only when its operand signals are available (either input or computed wires). Signal prioritisation may not be resolved if there exist two or more signals that depend on each other. This cyclic dependence indicates feedback, which violates a basic principle in the reversible paradigm.

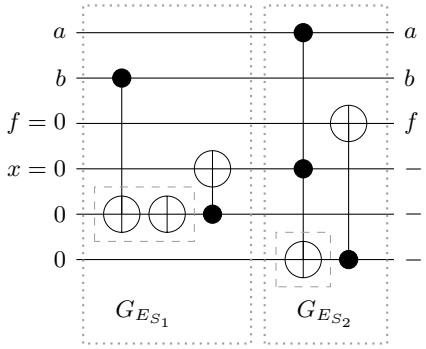


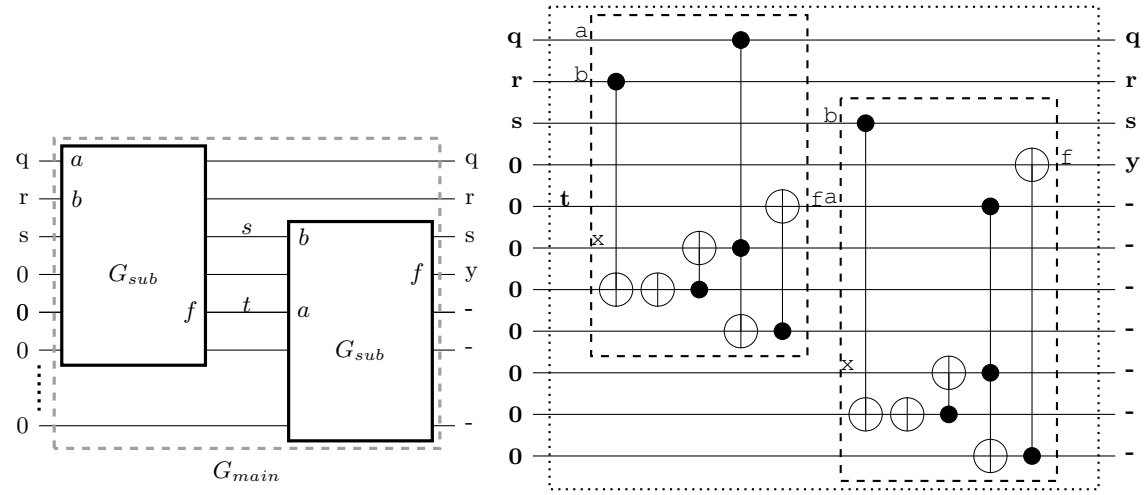
Figure 6.6: Reversible circuits realised using the VHDL code from Figure 6.3.

Example 34. *The VHDL code in Figure 6.3 contains two assignment statements, where each statement has an expression with one operator on its RHS. Consequently, two implicit lines are expected. The resulting circuit is shown in Figure 6.6.a.*

6.4.2 Components

As mentioned in Section 6.1, the structural style describes systems as a set of interconnected components. Components are entities instantiated within the architecture of another entity. Each instance places a sub-circuit definition within the main circuit. The reversible circuit of a component should be determined before interconnecting the main circuit in which it is instantiated. The order of the signal lines in the component circuit may differ from the signals mapped to them in the main circuit. Hence, a `port map` is associated with each instance to serve as a look-up table for line mapping.

Example 35. *Figure 6.2 shows a VHDL description of the entity* `main` *with a structural architecture body, which declares a component, then instantiates it twice. The component refers to the entity* `sub`*, as described in Figure 6.3, with the circuit G_{sub}, as shown in Figure 6.6. The structural interconnection of the main circuit G_{main} in Figure 6.7(a) follows the* `port map` *of each component instance to map the lines. This can be observed from Figure 6.7(b), where each instance sub-circuit is identical to the component circuit in Figure 6.6, but with lines rearranged according to the mapping of each instance. The circuit G_{main} is realised using **11** lines, **4** of which are ports (q, r, s, y), **1** internal signal t, and **6** implicit lines.*



(a) Structural interconnection of components (b) Circuit realisation on the gate level

Figure 6.7: The interconnecting structural VHDL architecture code from Figure 6.2.

6.5 Improving the Circuit Realisation

Expressions and other non-reversible actions implicitly add constant lines to the circuit. These lines are accumulated throughout statements and result in circuits with a large number of constant inputs, e.g., more than half of the lines in the circuit G_main computed in Example 35 are implicit. In this section, we propose line-aware arrangements to reduce the number of lines and gate costs.

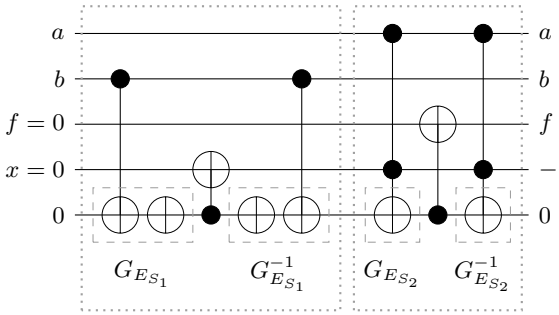


Figure 6.8: Reversible circuit realised using the VHDL code from Figure 6.3.

6.5.1 Line-aware Synthesis

According to the interconnection suggested in Section 6.2.1, implicit lines are assigned and used only once within the architecture body the outputs are garbage, i.e., not usable again in the circuit. Realising a statement with no garbage is possible when the RHS expression is computed in the reverse direction (re-computed). This technique was proposed for line-aware SyReC synthesis (see Section 3.3.1). More precisely, in addition to the two steps from Section 6.3, a third step is added:

3. Add the inverse circuit, G_E^{-1}, to re-compute the garbage.

The next statement reuses the same lines to realise a circuit with fewer lines.

Example 36. *Figure 6.8 shows the reversible circuit realisation for the VHDL code from Figure 6.3 following this scheme. The circuit requires only* 1 *implicit line instead of* 2 *lines as in the circuit from Figure 6.6.*

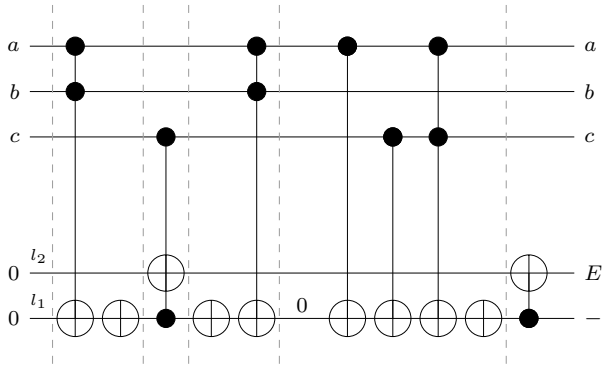


Figure 6.9: Line-aware realising of expression E from Example 33.

The ideas to realise optimised expressions in SyReC as proposed in Chapter 5 are applicable on VHDL by which expressions are computed with less constant inputs. Applying this optimised realisation on the expression from Example 33 results in the circuit shown in Figure 6.9, which is computed using **2** constant lines versus **6** in the direct realisation shown in Figure 6.4.

6.5.2 Gate-level Complexity Reduction

A constant input is not a signal fed into the circuit but it is like a literal numeric value in the code. In the conventional realisation of VHDL codes, numbers (i.e., literals) do not require circuits to compute their values as they are already specified in the code. Furthermore, an operation on a number operand can dramatically reduce the complexity of the circuit.

In the reversible circuit paradigm, a number is represented as a constant input. Using a constant input for each number in the code worsens the circuit parameters. On the other hand, considering constant signals, gate complexity may be reduced to lead towards optimised circuits[1]. This motivates exploiting the following two simple properties:

1. A control with a constant $'1'$ may be removed from the gate.

2. A Toffoli gate with one control known to be constant $'0'$ may be removed from the circuit.

[1]If the operation is applied to all number operands, then the circuit will be reduced to a set of constant inputs, i.e., no circuit at all.

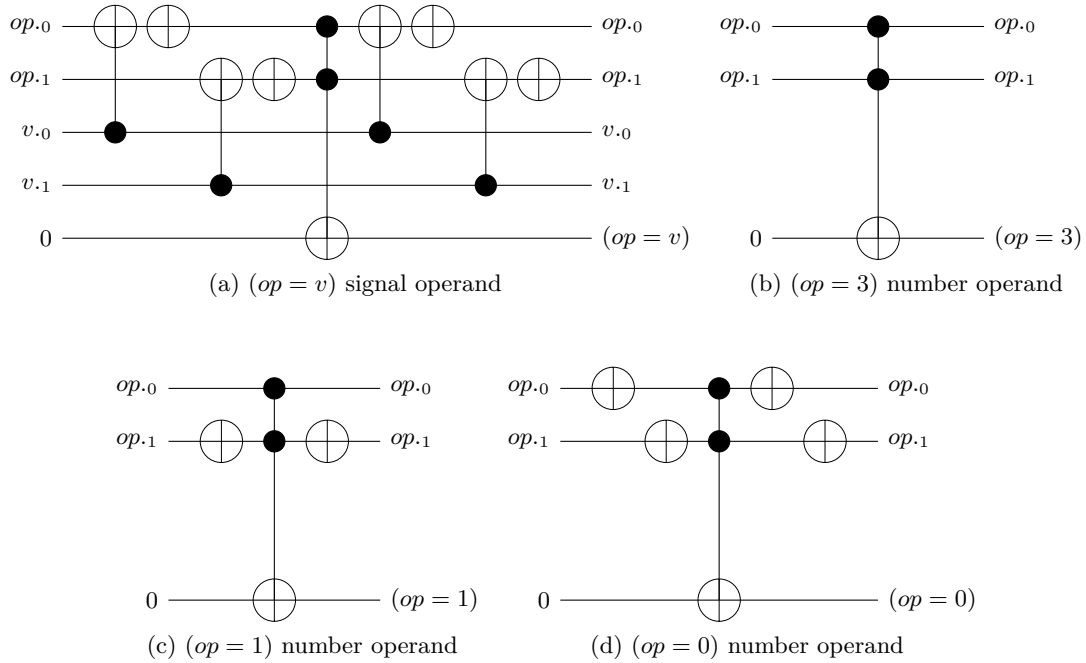


Figure 6.10: Gate-level optimisation of constant input.

Example 37. *Figure 6.10(a) shows the circuit of a 2-bit equality operation* `(op = v)` *where* `op` *and* `v` *are both variables. In the VHDL code of Figure 6.15, we can see a special case of this operation used as conditions, e.g.,* `(op = 0)`*. In this case, one of the operands is a constant number instead of a variable signal. Applying the complexity reduction rules, as suggested above, results in Figure 6.10(d), which uses fewer lines and has a lower gate cost. Applying the same optimisation on conditions with different numbers, such as* `(op = 3)` *and* `(op = 1)`*, results in a different circuits, e.g., Figures 6.10(b) and 6.10(c).*

6.6 Discussion

This section discusses the resulting VHDL-based synthesis approach for reversible circuits and compares it to the reversible-specific solution SyReC introduced in [24]. Between both solutions, one fundamental difference is the signal assignment, which is non-reversible in VHDL (<=, i.e., the previous signal value will be lost) and reversible in SyReC (^=, i.e., by additionally employing an XOR assignment, for example, which might require the addition of `out` and `wire` signals to realise the intended functionality). Also, the way to compute expressions, conditionals, and components are very similar between VHDL and SyReC when applied to constant inputs.

In the following, we consider two cases to study how these differences (and also the similarities) may affect the respectively obtained synthesis result.

6.6.1 Case Study: Gray-code to Binary-Code Conversion

Figure 6.11 shows a VHDL description[2]. of a 4-bit gray-code to binary code converter. The code defines two 4-bit vectors: `(g)` to the input gray-code and `(b)` for the output binary code. The architecture description in this code incorporates some repeated computations, e.g., `(g(3) xor g(2))` is computed three times. Hence, an equivalent description is proposed in Figure 6.12 to reduce the resulting computation complexity and the circuit cost. This code declares a 3-bit wire `w` to facilitate the computations. Despite being described using more statements, this code can be realised with reduced circuit costs, as shown in Table 6.1, which summarises the resulting circuit costs:

```
entity gray2binary is
  port (g : in STD_LOGIC_VECTOR (3 downto 0);
        b : out STD_LOGIC_VECTOR (3 downto 0));
end entity gray2binary;

architecture Behavioral of gray2binary is
begin
  b(3) <= g(3);
  b(2) <= g(3) xor g(2);
  b(1) <= g(3) xor g(2) xor g(1);
  b(0) <= g(3) xor g(2) xor g(1) xor g(0);
end behavioral;
```

Figure 6.11: 4-bit gray-code to binary converter using VHDL.

1. for basic realisation, as described in Sections 6.3 and 6.4, with no optimisation. This realisation results in lower gate count and cost, but with a larger number of lines.

2. for improved realisation as described in Section 6.5. This realisation shows higher gate count and cost, but with fewer lines.

```
architecture Behavioral of gray2binary is
  signal w (2 downto 0);
begin
  w(2) <=  g(3) xor g(2);
  w(1) <=  w(2) xor g(1);
  w(0) <=  w(1) xor g(0);
  b(3) <= g(3);
  b(2) <= w(2);
  b(1) <= w(1);
  b(0) <= w(0);
end architecture behavioral;
```

Figure 6.12: Optimized architecture description of Figure 6.11 to reduce complexity.

[2]This code is taken from `http://www.rfwireless-world.com`

We additionally consider a reversible HDL description, provided in SyReC syntax, as shown in Figure 6.13. This code conversion is reversible by definition, because of the one-to-one correspondence between the two codes. Hence, the gray-code to binary code converter is an ideal example to demonstrate the merits of SyReC-based synthesis (in its current state of development) compared to VHDL-based synthesis introduced above. The SyReC approach performs significantly better because reversibility can fully be exploited.

```
module gray2binary(inout x(4))
        x.2 ^= x.3
        x.1 ^= x.2
        x.0 ^= x.1
```

Figure 6.13: Gray-code to binary code converter using SyReC.

Table 6.1: Experimental results of the 4-bit gray-code to binary code converter.

Parameter	VHDL				SyReC
	Figure 6.11		Figure 6.12		Figure 6.13
	1	2	1	2	
Gates	16	28	10	16	**3**
Total lines	14	11	11	9	**4**
Quantum cost	16	28	10	16	**3**
Transistor cost	128	224	80	128	**24**

6.6.2 Case Study: Logic Unit

As another example, we consider a case where non-reversible functionality should be realised. Figures 6.14 and 6.15 show two equivalent codes describing a 32-bit logic-unit described in SyReC[3] and VHDL, respectively. A conditional assignment computes the output signal `x0`. In the SyReC code, `x0` is initialized using the XOR-operator (`^=`), e.g., in `x0 ^= (x1 & x2)`. Here, the operation is identical to (<=) in VHDL since `x0` is an `out` signal.

For the SyReC code, the circuit is realised in four different configurations as summarised in Table 5.1, namely:

1. The basic SyReC synthesis: incorporates no optimisation and results in a low gate count [24].

2. The line-aware synthesis: implements statement re-compute and reuses circuit lines and exploits fewer lines [62].

3. The cost-aware synthesis: exploits an extra (helper) line to reduce gates cost, and this configuration shows lower cost measures [42].

[3] A SyReC benchmark (**lu_238.src**) in *RevLib* [67].

```
1          module lu(in op(2), out x0, inout x1, inout x2)
2            if (op = 0) then
3              x0 ^= (x1 & x2)
4            else
5              if (op = 1) then
6                x0 ^= (x1 | x2)
7              else
8                if (op = 2) then
9                  x0 ^= (x1 ^ x2)
10               else
11                 x0 ^= x1
12                 ~= x0
13               fi (op = 2)
14             fi (op = 1)
15           fi (op = 0)
```

Figure 6.14: A SyReC description of a basic 32-bit arithmetic unit *lu_238.src*.

4. Optimum trade-off: enables the re-compute option, as in configuration (2), as well as a helper line option, as in configuration (3), and resulted in a compromised circuit with the best trade-off between lines and cost metrics.

On the other hand, the VHDL-code is synthesised using:

1. the basic realisations as described in Sections 6.3 and 6.4, which results in the lowest costs compared to all other experiments, but with a large number of lines.

2. the improved realisation as described in Section 6.5, which requires a number of lines equals to the best SyReC experiment while, at the same time, it shows a low-cost measure, as compared to all SyReC experiments.

```
1 entity alu is
2   port (op   :  in unsigned (1 downto 0);
3         x1,x2:  in bit_vector (31 downto 0);
4         x0   :  out bit_vector (31 downto 0));
5 end entity test;
6
7 architecture data_flow of lu is
8 begin
9   x0 <= (x1 and x2) when (op = 0) else
10        (x1 or  x2) when (op = 1) else
11        (x1 xor x2) when (op = 2) else
12        (not x1);
13 end architecture data_flow;
```

Figure 6.15: A VHDL description of a basic 32-bit arithmetic unit.

The results in Table 6.2 show that VHDL could compete or even overtake SyReC when it comes to non-reversible functions. So, the two cases show that HDL efficiency is highly problem-dependent. As VHDL is not able to compete with SyReC in realising reversible functions, e.g., shown by the code-conversion case, it can still realise arbitrary (non-reversible)

functions with comparable or even better circuits, e.g., as shown by the logic-unit case. In either case, VHDL is more convenient for designers with no or little knowledge of the reversible computation paradigm.

Table 6.2: Experimental results of a 32-bit logic unit.

Parameter	SyReC				VHDL	
	1	2	3	4	1	2
Gates	**384**	612	392	622	414	671
Total lines	197	**133**	198	134	235	**133**
Quantum cost	6557	10462	**2312**	3894	**682**	1207
Transistor cost	9856	15616	**6360**	10288	**3472**	5752

6.7 Summary

In this chapter, we considered a conventional hardware description language (VHDL) to realise reversible circuits. The structural description model is analysed and direct realisations for signals, expressions, assignments, conditionals, and components are proposed.

Optimised realisations are also proposed for higher quality circuits, including a line-aware synthesis and a complexity reduction arrangement. The differences and similarities were discussed as compared to the dedicated reversible HDL (SyReC). The discussion shows that each approach has some advantages and declaring a winner depends on the problem to be described.

The chapter provides an elementary basis towards a conventional HDL-based reversible circuit design, with only a little knowledge for the programmer in this computational paradigm.

Chapter 7

Improving the Grammar of SyReC

To be accepted as a first-choice tool for reversible circuit design, SyReC programming should be convenient for a wide range of programmers along with its scalability, modularity, and the quality of its circuits. The previous efforts invested in improving the SyReC design approach were focused on synthesis-related issues aimed a realising higher-quality circuits. So, language's grammar has not been revised since its first release in 2010 [61]. With our experience in SyReC programming, we highlight possible enhancements and extensions to the grammar of this HDL to make the language more powerful and more convenient as well. In this chapter, we propose grammatical extensions to SyReC by appending some syntactical rules, modifying others, and defining more data operations.

7.1 Control Logic

To guarantee reversibility of the descriptions of SyReC, a reversible control flow is implemented [24]. Consequently, conditional statements do require not only an *if-condition* (to decide which of the then- or the else-block is to be executed next) but also a *fi-condition* (for the same reason, if the computation is conducted in reverse direction). This was first introduced in the reversible software language *Janus* [64], where the fi-condition is called an assertion. So, SyReC, which is based on Janus, inherited this assertion. Moreover, HDL descriptions do occur from which it is not possible to realise a reversible control flow. Hence, designers of reversible circuits and systems are faced with the problem of properly describing a reversible control flow and the uncertainty whether such a control flow is even possible. With this violation, it is not apparent whether the entire SyReC description is fully reversible.

Example 38. *The if-condition of the Figure 7.1(a) is* `(a = 5)` *and the signal* `a` *is not updated within the statement. Therefore the same expression is valid as a fi-condition and the statement is fully reversible. On the other hand, the if-condition in Figure 7.1(b) is* `(x = y)`*. Here, the signals used to compute the expression are updated within the statement, and so the same expression is no longer a valid fi-condition. To incorporate signal updates, a correspondingly adjusted fi-condition is required,* `((x - 1) = y)` *in this case. This*

```
if (a = 5)
then
  x += 1
else
  y += 2
fi (a = 5)
```

(a) Fully reversible if-statement

```
if (x = y)
then
  x += 1
else
  y += 2
fi ((x - 1) = y)
```

(b) Partially reversible if-statement

Figure 7.1: Reversibility in SyReC conditional statements.

modified expression works for most of the possible assignments of x and y in both directions. However, a problem occurs if, e.g., x = 4 and y = 1 are considered. In the forward direction, this would not satisfy the if-condition and would trigger the execution of the else-block (leading to x = 4 and y = 3). This assignment, however, would satisfy the fi-condition, i.e., if executed in reverse direction, the then-block would reversibly be executed (leading to x = 3 and y = 3). In other words, the two input states (x, y) = (4, 1) and (x, y) = (3, 3) both map to the output state (4, 3), which is a clear violation of the reversible computing paradigm. Such statements are referred to as being partially-reversible.

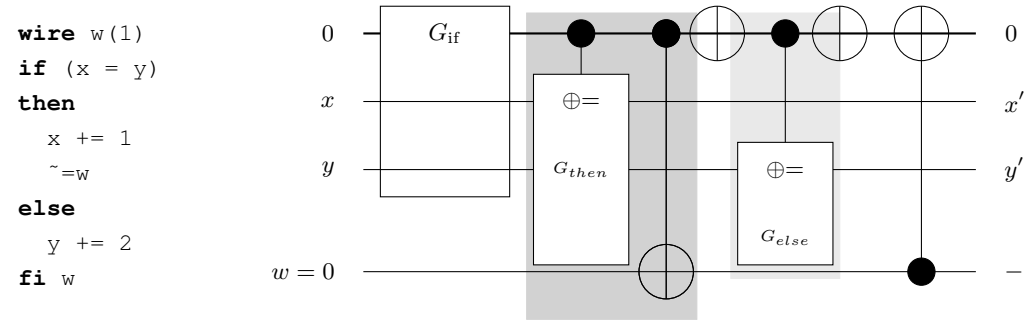
An if-statement is partially reversible, if there exist two different input states with an execution of statements that yields the same output state. Generating a fi-condition or checking for full reversibility is not always an intuitive task. This issue is addressed in [71] in which automated generation of fi-expressions use *predicate transformer semantics*, that are based on *Hoare logic* [72]. Checking whether a given reversible SyReC description indeed is fully reversible is another challenge for which the designers must to be aware. This partially-reversible description can be checked as well [71]. When a fi-condition is generated and successfully passes the full-reversibility check, an explicit fi-condition is no longer necessary, as it can be automatically generated.

7.1.1 Implicit fi-conditions

Realising a fully reversible if-statement follows the usual scheme as in Figure 3.13(b). However, when a code is identified as being partially-reversible, the question remains how to fix the problem, i.e., how to transfer this description into a reversible one. In fact, it is not always possible or desirable to generate a fully-reversible condition.

As an alternative, a simple programming hack can solve this problem. Here, we accept this partial reversibility and, instead, apply additional circuit lines. More precisely, an additional 1-bit wire signal is applied with an initial default value of 0 and is set to 1 if and only if the if-condition is satisfied. Then, this signal is used to trigger either the respective realisation of the then-block or the else-block as shown in Figure 7.2. This solution guarantees a correct computation with a penalty of only a single bit.

However, to make this solution consistent with the definition of SyReC (where partially



(a) fi-condition using a wire (b) Realisaion of a partially-reversible fi-condition

Figure 7.2: Fully-reversible fi-condition using an internal wire.

reversible descriptions are not allowed), we must modify the grammar. In fact, we simply make the fi-condition optional as shown in Figure 7.8 (line 12). Whenever the fi-condition is provided, a synthesiser simply realises the corresponding expression. Whenever the condition is omitted, a synthesiser uses the algorithms proposed in [71] to generate a suitable fi-condition. If this is possible (i.e., the conditional statement is fully-reversible), then the resulting fi-condition is realised. Otherwise, a single bit wire is automatically declared and applied as described above.

7.1.2 Reversible Case-statement

Considering the same approach in dealing with fi-conditions, other control statements can be proposed, and a reversible case-statement is now possible to be realized. Case-statements represent a special form to represent nested if-statement structures where all if-conditions have the same form `(choice_expression = <number>)`. Instead of repeating the choice expression with each if-condition, it is provided once at the beginning of the case statement. This will tangibly enhance the readability of the code and simplify the structure. It is an intuitive task to map these case-statements into equivalent nested-if structures and does not constitute a serious obstacle for the synthesis process. Overall, this motivates extensions to the SyReC grammar as shown in Figure 7.8 (lines 28 and 30).

Example 39. *Figures 7.3(a) and 7.3(b) show two equivalent SyReC specifications where the former is written using the original SyReC grammar (i.e., as a cascade of conditional statements) and the latter is written using the proposed case statement.*

7.2 Data Operations

Reversible HDLs are supposed to facilitate the description of reversible circuits. This includes a complete set of defined data-operations. The current version of SyReC already provides many data operations, but misses essential operations such as bit-wise rotation operations, which is a defined operation in other HDLs, such as VHDL. The operation is

```
1  module simple_alu(in op(2), in a, in b, out c)
2          if (op = 0)
3          then
4            c ^= (a + b)
5          else
6            if (op = 1)
7            then
8              c ^= (a - b)
9            else
10             if (op = 2)
11             then
12               c ^= (a * b)
13             else
14               c ^= (a / b)
15             fi (op = 2)
16           fi (op = 1)
17         fi (op = 0)
```

(a) Nested if-fi code

```
1  module simple_alu(in op(2), in a, in b, out c)
2          with  op select
3            case 0:       c ^= (a + b)
4            case 1:       c ^= (a - b)
5            case 2:       c ^= (a * b)
6            case default: c ^= (a / b)
7          endcase
```

(b) Reversible case statement

Figure 7.3: SyReC description of a simple arithmetic unit.

obviously reversible, and it receives special significance in cyclic-coding applications and cryptography [73].

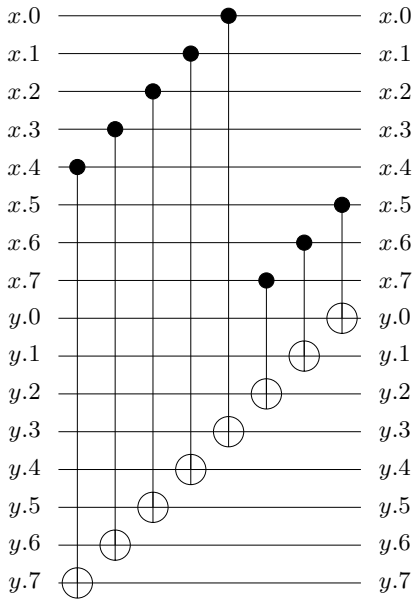
In the original grammar of SyReC, programmers needed to write a sophisticated code to perform a rotation operation bit-by-bit. To accomplish this, they could use the related shift operators << and >> or XOR assignment statements ^=, which are not reversible and require additional circuit lines. This is not only counter-intuitive and against the aim of facilitating the design process, but also yields significantly larger circuits.

Example 40. *Figure 7.4 shows a SyReC code to update the 8-bit signal* y *using the* ^= *operator. The value of signal* x *is rotated 3-bits to the left of the bit-wise and XOR-ed with* y*. This code shows that the operation is defined bit-by-bit. The resulting code is not intuitive. Moreover, the circuit is large due to the need to embed this shift operation with an additional signal* x *(see Figure 7.4(b)).*

SyReC grammar is extended such that bit-wise rotation is incorporated. The operations <| and |> are defined for rotate-left and rotate-right, respectively. Accordingly, the rotation

```
module test(inout x(8), inout y(8))
  y.7:3 ^= x.4:0
  y.2:0 ^= x.7:5
```

(a) Bit-wise rotation



(b) Circuit realisation of program

Figure 7.4: Original SyReC description realising a rotation.

in Figure 7.4.a can be replaced by the statement `y ^= (x <| 3)`

Since a rotation operation only swaps bits by a respectively given number, both newly added operations can be synthesised by a set of swap statements, without any constant input. This implementation allows for a reversible signal update to be defined.

Example 41. *Figure 7.5(a) is a code to rotate signal* `y` *by 3-bits to the left. The code is realised using SWAP gates, as shown in Figure 7.5(b), where the rotation update the signal without any other line being involved. This compact operation is described using the reversible update statement* `y <|= 3`*, according to the revised grammar.*

For the above operations to be incorporated in SyReC, we modified the grammar, as shown in Figure 7.8 (lines 9, 16, 23).

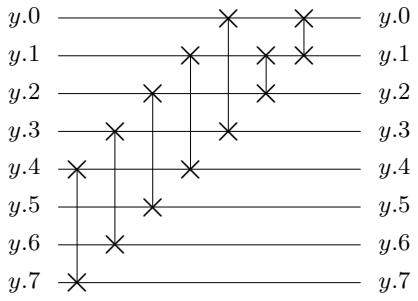
7.3 Import of Alternative Circuit Descriptions

SyReC is a modular language, where the program is defined as a group of modules, and the main of which is the top-level module composed of statements and sometimes other modules. When SyReC parses the code, it generates a tree-structure for the main module, which contains sub-modules as sub-trees.

The next phase is to convert this tree-structure into a reversible circuit. A terminals (leaf) is the smallest entity in the tree, which has a reversible circuit defined. Larger entities interconnect these circuits together step-by-step until the main circuit is computed. When

```
module test(inout x(8), inout y(8))
  y.7:3 <=> y.4:0
  y.2:1 <=> y.1:0
```

(a) Bit-wise rotation



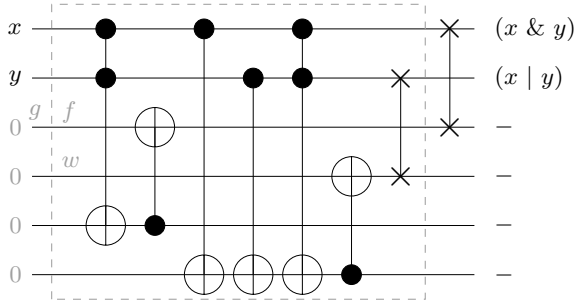
(b) Circuit realisation

Figure 7.5: SyReC description realizing of a signal rotation.

```
module sub_circuit(inout a(1), inout b(1), out f(1))
  wire w(1)
  f ^= (a & b)
  w ^= (a | b)
  w <=> b

module main(inout x(1), inout y(1))
  wire g(1)
  call sub_circuit(x,y,g)
  x <=> g
```

(a) Multiple module code



(b) Reversible circuit realisation

Figure 7.6: SyReC description with sub-modules.

a module is called by another module (main), its circuit must be computed first; then it is used to compute the main circuit.

Example 42. *Consider Figure 7.6(a), which shows a SyReC program composed of two modules* `main` *and* `sub_circuit`*. The first is the top-level module that calls the later in line 9 using a call statement. Applying the synthesis method, this yields the circuit as shown the bottom of Figure 7.6(b).*

This modularity has only been used to call circuit descriptions provided in the SyReC language itself. However, we may have better circuits than those generated by SyReC synthesised using different methodologies, but the current version of SyReC cannot use such circuits. This frequently prevents the realisation of more compact circuits because often cheaper building blocks can not be described in SyReC, but only in terms of gate level descriptions.

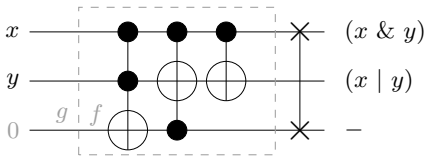
To avoid this problem, we propose an extension to support the import of alternative circuit descriptions (as an example, circuits described in the (`.real`) file format as introduced in [67] are considered). So, we extend the grammar from Figure 3.1 by allowing an `<import-list>` as defined in Figure 7.8 (lines 1, 2, and 29). This arrangement does

not require additional complexity to the circuit synthesis as it simplifies the computation since some parts of the circuit are already determined, and all we need is into substitute (interconnect) them in the main circuit.

```
import sub_circuit from circuit_file.real

module main(inout x(1), inout y(1))
  wire g(1)
  call sub_circuit(x,y,g)
  x <=> g
```

(a) SyReC program with circuit import



(b) Reversible circuit realisation

```
# ----- circuit_file.real ------
# This file has been generated using
# RevKit 1.3-snapshot (www.revkit.org)
.version        2.0
.numvars        3
.variables      x0 x1 x2
.inputs         a b const_0
.outputs         a b f
.constants      --0
.garbage        --1
.begin
t3 x0 x1 x2
t3 x0 x2 x1
t2 x0 x1
.end
```

(c) Circuit netlist(.real file format)

Figure 7.7: SyReC description and realisation with imported circuit.

Example 43. *Figure 7.7(a) shows a circuit description which is functionally equivalent to the description considered before in Figure 7.6(a). However, instead of describing the SyReC module* `sub_circuit`*, the corresponding reversible circuit is provided as shown in Figure 7.7(b)*[1] *and imported using the newly-added statement (line 1). Overall, this yields a significantly smaller circuit as shown in Figure 7.7(c).*

According to the concept of modularity, which is based on information hiding, the details of each module are irrelevant to the other modules. Consequently, importing circuits is not supposed to change the call statement, so it does not have any effect on the code of the main module. However, because the functions of imported circuits are not explicitly described in the code, it is necessary to verify the signal definition before importing a circuit. Specifically, it is important to verify that input and output specifications of the imported circuit match the corresponding signals in the SyReC module, e.g., constant inputs and bit-widths of signals ,

[1] *RevLib* uses the (*.real*) file format to save netlists of reversible circuits [67].

for which it would be more convenient to import entities described using another HDL, such as VHDL, that utilises port maps to explicitly specify signal types and bit-widths. With this modification it becomes possible to achieve a hybrid HDL specification described using both SyReC and VHDL, where SyReC modules call VHDL entities.

7.4 Summary

In this chapter, SyReC grammar is revised for the first time. The original grammar causes several shortcomings making it harder to describe the desired behaviour and often yields more expensive circuits. These shortcomings are introduced with extensions to the language and a proposed revised grammar (shown in Figure 7.8) that includes:

1. solving possible violations of full reversibility caused by fi-conditions in some if-then-else-fi statements (known as partial reversibility). Moreover, the proposed realisation of partially-reversible statements allows for a reversible `case` statement. Both modifications simplify the grammar for more convenient SyReC programming.

2. defining new operations on data, such as a bit-wise rotation, which is a reversible operation. Hence, the modified grammar defines these operations within expressions and as a reversible signal update statement.

3. enabling hybrid design by reusing circuit definitions, synthesised possibly using methodologies other than SyReC. The required circuits are imported and then connected to the main circuit using a normal call statement. This opens a door for the integration of different methodologies within one design flow, which can result in circuits with higher qualities.

Program and Modules

1 ⟨*program*⟩ ::= [⟨*import-list*⟩] ⟨*module*⟩ {⟨*module*⟩}
2 ⟨*import-list*⟩ ::= **'import'** ⟨*identifier*⟩ **'from'** ⟨*file*⟩ {**','** ⟨*identifier*⟩ **'from'** ⟨*file*⟩ }
3 ⟨*module*⟩ ::= **'module'** ⟨*identifier*⟩ **'('** [⟨*parameter-list*⟩] **')'** {⟨*signal-list*⟩} ⟨*statement-list*⟩
4 ⟨*parameter-list*⟩ ::= ⟨*parameter*⟩ {**','** ⟨*parameter*⟩}
5 ⟨*parameter*⟩ ::= (**'in'** | **'out'** | **'inout'**) ⟨*signal-declaration*⟩
6 ⟨*signal-list*⟩ ::= (**'wire'** | **'state'**) ⟨*signal-declaration*⟩ {**','** ⟨*signal-declaration*⟩}
7 ⟨*signal-declaration*⟩ ::= ⟨*identifier*⟩ {**'['**⟨*int*⟩**']'**} [**'('**⟨*int*⟩**')'**]

Statements

8 ⟨*statement-list*⟩ ::= ⟨*statement*⟩ {**';'** ⟨*statement*⟩}
9 ⟨*statement*⟩ ::= ⟨*call-statement*⟩ | ⟨*for-statement*⟩ | ⟨*if-statement*⟩ | ⟨*unary-statement*⟩ | ⟨*assign-statement*⟩ | ⟨*swap-statement*⟩ | ⟨*skip-statement*⟩ | ⟨*case-statement*⟩| ⟨*rotate-statement*⟩
10 ⟨*call-statement*⟩ ::= (**'call'** | **'uncall'**) ⟨*identifier*⟩ **'('** (⟨*identifier*⟩ {**','** ⟨*identifier*⟩}) **')'**
11 ⟨*for-statement*⟩ ::= **'for'** [[**'$'** ⟨*identifier*⟩ **'='**] ⟨*number*⟩ **'to'**] ⟨*number*⟩ [**'step'** [**'-'**] ⟨*number*⟩] ⟨*statement-list*⟩ **'rof'**
12 ⟨*if-statement*⟩ ::= **'if'** ⟨*expression*⟩ **'then'** ⟨*statement-list*⟩ **'else'** ⟨*statement-list*⟩ **'fi'** [⟨*expression*⟩]
13 ⟨*case-statement*⟩ ::= **'with'** ⟨*identifier*⟩ **'select'** { **'case'** ⟨*number*⟩ **':'** ⟨*statement-list*⟩} [**'case'** **'default'** **':'** ⟨*statement-list*⟩] **'endcase'**
14 ⟨*assign-statement*⟩ ::= ⟨*signal*⟩ (**'^'** | **'+'** | **'-'**) **'='** ⟨*expression*⟩
15 ⟨*unary-statement*⟩ ::= (**'~'** | **'++'** | **'--'**) **'='** ⟨*signal*⟩
16 ⟨*rotate-statement*⟩ ::= ⟨*signal*⟩ (**'<|'** | **'|>'**) **'='** ⟨*number*⟩
17 ⟨*swap-statement*⟩ ::= ⟨*signal*⟩ **'<=>'** ⟨*signal*⟩
18 ⟨*skip-statement*⟩ ::= **'skip'**
19 ⟨*signal*⟩ ::= ⟨*identifier*⟩ {**'['** ⟨*expression*⟩ **']'**} [**'.'** ⟨*number*⟩ [**':'** ⟨*number*⟩]]

Expressions

20 ⟨*expression*⟩ ::= ⟨*number*⟩ | ⟨*signal*⟩ | ⟨*binary-expression*⟩ | ⟨*unary-expression*⟩ | ⟨*shift-expression*⟩
21 ⟨*binary-expression*⟩ ::= **'('** ⟨*expression*⟩ (**'+'** | **'-'** | **'^'** | **'*'** | **'/'** | **'%'** | **'*>'** | **'&&'** | **'||'** | **'&'** | **'|'** | **'<'** | **'>'** | **'='** | **'!='** | **'<='** | **'>='**) ⟨*expression*⟩ **')'**
22 ⟨*unary-expression*⟩ ::= (**'!'** | **'~'**) ⟨*expression*⟩
23 ⟨*shift-expression*⟩ ::= **'('** ⟨*expression*⟩ (**'<<'** | **'>>'**| **'<|'** | **'|>'**) ⟨*number*⟩ **')'**

Identifier and Constants

24 ⟨*letter*⟩ ::= (**'A'** | ... | **'Z'** | **'a'** | ... | **'z'**)
25 ⟨*digit*⟩ ::= (**'0'** | ... | **'9'**)
26 ⟨*identifier*⟩ ::= (**'_'** | ⟨*letter*⟩) {(**'_'** | ⟨*letter*⟩ | ⟨*digit*⟩)}
27 ⟨*int*⟩ ::= ⟨*digit*⟩ {⟨*digit*⟩}
28 ⟨*number*⟩ ::= ⟨*int*⟩ | **'#'** ⟨*identifier*⟩ | **'$'** ⟨*identifier*⟩ | (**'('** ⟨*number*⟩ (**'+'** | **'-'** | **'*'** | **'/'**) ⟨*number*⟩ **')'**)
29 ⟨*file*⟩ ::= ⟨*identifier*⟩ [**'.real'**]

Figure 7.8: The modified syntax of the hardware description language SyReC.

Chapter 8

Conclusions

The traditional technology of computing machines developed over the last decades is about to reach its limits. Consequently, an alternative paradigm needs to evolve sooner to replace the conventional paradigm. Reversible computation may provide such an alternative replacement or, at least, an enhancement of computing machines. Reversible logic is highly-relevant to some interesting applications, such as quantum computing, low-power design, adiabatic circuits, and encoding and decoding devices.

Different methodologies were proposed to realise functions as reversible circuits. Most do not scale well, i.e., they are not capable of handling large design problems. One approach to achieve scalability through hardware description languages, which are corner stones of the design flow of conventional circuits. Similarly, the HDL approach is also considered for reversible circuit design. A dedicated HDL, SyReC, is introduced for describing reversible circuits, and it shows the capacity to describe, with simple codes, relatively large design problems, which are beyond the capacity of any other approach. The major drawback of SyReC is that it realises reversible circuits with a large number of lines.

In this dissertation, an HDL-based design approach is optimised to generate reversible circuits with fewer lines. This includes improvements at different design levels, which are made without compromising the advantages of the design approach, i.e., scalability and simplicity.

The first improvement described in Chapter 4, is made on the programming style of SyReC. Programmers, which are typically influenced by the conventional description style, deal with different paradigm considerations. The synthesis flow was investigated, and rules for SyReC programming with more efficiency are proposed resulting in circuits that are synthesized with tangibly better metrics. However, SyReC codes written according to the proposed style are still less readable as compared to the spontaneous style.

In Chapter 5 we introduce line-aware realisations of SyReC expressions with which we avoid rewriting codes according to a certain programming style. Expressions in SyReC do not assume operations are reversible, so the realisations of these expressions follow conventional computations. We exploit properties of reversible computations in realising circuits with better metrics for SyReC expressions. The fact that expressions in SyReC are not assuming

reversibility at all in its operations makes the concept introduced in this chapter applicable for SyReC expressions as well as in conventional HDLs.

In Chapter 6 we investigate the conventional hardware description language, VHDL. We propose realisations of structural descriptions for architectures of entities in VHDL to introduce a basis for a design flow that reuses elaborated conventional circuit design flow and its efficient tools and requires less knowledge in reversible paradigm. This may encourage stakeholders to accept reversible circuits as a practical alternative.

In Chapter 7 we propose some syntactical modifications on SyReC grammar including new control statements and definitions of new operations. Another modification allows SyReC to import circuits that are designed using different methods and integrate them within the modules. This enables a hybrid design that profits from the advantages of each approach.

Finally, HDL-based design of reversible circuits cannot achieve synthesis with minimal lines, but it can at least approach this goal within the scope of this dissertation. Also, reaching a thorough design flow for a reversible circuit, which can be considered as a practical alternative, needs additional time to be well-established. Upcoming challenges are expected, and more effort is required. This dissertation contributes as an important step towards an elaborated design flow of reversible circuits.

Bibliography

[1] G. E. Moore, "Cramming more components onto integrated circuits," *Electronics*, vol. 38, no. 8, pp. 114–117, 1965.

[2] C. H. Bennett, "Logical reversibility of computation," *IBM J. Res. Dev*, vol. 17, no. 6, pp. 525–532, 1973.

[3] V. V. Shende, A. K. Prasad, I. L. Markov, and J. P. Hayes, "Synthesis of reversible logic circuits," *IEEE Trans. on CAD*, vol. 22, no. 6, pp. 710–722, 2003.

[4] D. Große, R. Wille, G. W. Dueck, and R. Drechsler, "Exact multiple control Toffoli network synthesis with SAT techniques," *IEEE Trans. on CAD*, vol. 28, no. 5, pp. 703–715, 2009.

[5] R. Wille and R. Drechsler, "Synthesizing reversible logic: An overview," in *Int'l Workshop on Applications of the Reed-Muller Expansion in Circuit Design*, pp. 79–86, 2009.

[6] N. Abdessaied and R. Drechsler, *Reversible and Quantum Circuits Optimization and Complexity Analysis*. Springer, 2016.

[7] G. F. Viamontes, M. Rajagopalan, I. L. Markov, and J. P. Hayes, "Gate-level simulation of quantum circuits," in *ASP Design Automation Conf.*, pp. 295–301, 2003.

[8] R. Wille, D. Große, D. M. Miller, and R. Drechsler, "Equivalence checking of reversible circuits," in *Int'l Symp. on Multi-Valued Logic*, pp. 324–330, 2009.

[9] R. Landauer, "Irreversibility and heat generation in the computing process," *IBM journal of research and development*, vol. 5, no. 3, pp. 183–191, 1961.

[10] A. Berut, A. Arakelyan, A. Petrosyan, S. Ciliberto, R. Dillenschneider, and E. Lutz, "Experimental verification of Landauer's principle linking information and thermodynamics," *Nature*, vol. 483, pp. 187–189, 2012.

[11] B. Desoete and A. D. Vos, "A reversible carry-look-ahead adder using control gates," *INTEGRATION, the VLSI Jour.*, vol. 33, no. 1-2, pp. 89–104, 2002.

[12] P. Patra and D. Fussell, "On efficient adiabatic design of MOS circuits," in *Workshop on Physics and Computation*, (Boston), pp. 260–269, 1996.

[13] R. Wille, R. Drechsler, C. Oswald, and A. Garcia-Ortiz, "Automatic design of low-power encoders using reversible circuit synthesis," in *Proceedings of the Conference on Design, Automation and Test in Europe*, pp. 1036–1041, EDA Consortium, 2012.

[14] M. Nielsen and I. Chuang, *Quantum Computation and Quantum Information*. Cambridge Univ. Press, 2000.

[15] R. Glück and M. Kawabe, "A method for automatic program inversion based on LR(0) parsing," *Fundamenta Informaticae*, vol. 66, pp. 367–395, 01 2005.

[16] S. Abramov and R. Glück, "The universal resolving algorithm and its correctness: inverse computation in a functional language," *Science of Computer Programming*, vol. 43, no. 2âĂŞ3, pp. 193 – 229, 2002.

[17] N. Weste and D. Harris, *CMOS VLSI Design a Circuits and Systems Perspective*. Addison-Wesley, 4th ed., 2010.

[18] P. J. Ashenden, *The Designers Guide to VHDL the Science of Microfabrication*. Elsevier, 3rd ed., 2008.

[19] R. Nane, V. M. Sima, C. Pilato, J. Choi, B. Fort, A. Canis, Y. T. Chen, H. Hsiao, S. Brown, F. Ferrandi, J. Anderson, and K. Bertels, "A survey and evaluation of fpga high-level synthesis tools," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, pp. 1591–1604, Oct 2016.

[20] R. Wille, M. Soeken, D. M. Miller, and R. Drechsler, "Trading off circuit lines and gate costs in the synthesis of reversible logic," *INTEGRATION, the VLSI Jour.*, vol. 47, no. 2, pp. 284–294, 2014.

[21] IEEE Std. 1076-2008, *IEEE Standard VHDL Language Reference Manual*, 2019.

[22] IEEE Std. 1346-2005, *IEEE Standard Verilog hardware description language*, 2006.

[23] IEEE Std. 1800-2012, *IEEE Standard for SystemVerilog–Univied Hardware Design, Specification, and Verification Language*, 2013.

[24] R. Wille, E. Schönborn, M. Soeken, and R. Drechsler, "SyReC: A hardware description language for the specification and synthesis of reversible circuits," *INTEGRATION, the VLSI Jour.*, vol. 53, pp. 39–53, 2016.

[25] R. Wille, M. Soeken, D. Große, E. Schönborn, and R. Drechsler, "Designing a RISC CPU in reversible logic," in *Int'l Symp. on Multi-Valued Logic*, pp. 170–175, 2011.

[26] Z. Alwardi, R. Wille, and R. Drechsler, "Towards line-aware realizations of expressions for HDL-based synthesis of reversible circuits," in *Reversible Computation*, pp. 233–247, Springer International Publishing, 2015.

[27] Z. Alwardi, R. Wille, and R. Drechsler, "Re-writing HDL descriptions for line-aware synthesis of reversible circuits," in *Int'l Symp. on Multi-Valued Logic*, pp. 31–36, IEEE, 2016.

[28] Z. Alwardi, R. Wille, and R. Drechsler, "Extensions to the reversible hardware description language SyReC," in *Int'l Symp. on Multi-Valued Logic*, pp. 185–190, IEEE, 2017.

[29] Z. Alwardi, R. Wille, and R. Drechsler, "Towards VHDL-based design of reversible circuits reversible circuits," in *Reversible Computation*, pp. 102–108, Springer International Publishing, 2017.

[30] Z. Alwardi, R. Wille, and R. Drechsler, "Synthesis of reversible circuits using conventional hardware description languages," in *Int'l Symp. on Multi-Valued Logic*, IEEE, 2018.

[31] T. Sasao, "Logic synthesis with EXOR-gates," in *Logic Synthesis and Optimization* (T. Sasao, ed.), pp. 259–285, Kluwer Academic Publisher, 1993.

[32] R. E. Bryant, "Graph-based algorithms for Boolean function manipulation," *IEEE Trans. on Comp.*, vol. 35, no. 8, pp. 677–691, 1986.

[33] M. Soeken, R. Wille, O. Keszocze, D. M. Miller, and R. Drechsler, "Embedding of large Boolean functions for reversible logic," *ACM Journal on Emerging Technologies in Computing Systems*, 2015, accepted.

[34] D. M. Miller, R. Wille, and G. Dueck, "Synthesizing reversible circuits for irreversible functions," in *EUROMICRO Symp. on Digital System Design*, pp. 749–756, 2009.

[35] R. Wille, O. Keszöcze, and R. Drechsler, "Determining the minimal number of lines for large reversible circuits," in *Design, Automation and Test in Europe*, 2011.

[36] D. Maslov and G. W. Dueck, "Reversible cascades with minimal garbage," *IEEE Trans. on CAD*, vol. 23, no. 11, pp. 1497–1509, 2004.

[37] T. Toffoli, "Reversible computing," in *Automata, Languages and Programming* (W. de Bakker and J. van Leeuwen, eds.), p. 632, Springer, 1980. Technical Memo MIT/LCS/TM-151, MIT Lab. for Comput. Sci.

[38] E. F. Fredkin and T. Toffoli, "Conservative logic," *International Journal of Theoretical Physics*, vol. 21, no. 3/4, pp. 219–253, 1982.

[39] M. Saeedi and I. Markov, "Synthesis and optimization of reversible circuits-a survey," *ACM Computing Surveys*, vol. 45, no. 2, p. 21, 2013.

[40] V. V. Zhirnov, R. K. Cavin, J. A. Hutchby, and G. I. Bourianoff, "Limits to binary logic switch scaling – a gedanken model," *Proc. of the IEEE*, vol. 91, no. 11, pp. 1934–1939, 2003.

[41] R. Wille, M. Soeken, and R. Drechsler, "Reducing the number of lines in reversible circuits," in *Design Automation Conference*, pp. 647–652, IEEE, 2010.

[42] D. M. Miller, R. Wille, and R. Drechsler, "Reducing reversible circuit cost by adding lines," in *Int'l Symp. on Multi-Valued Logic*, pp. 217–222, 2010.

[43] M. K. Thomson and R. Glück, "Optimized reversible binary-coded decimal adders," *J. of Systems Architecture*, vol. 54, pp. 697–706, 2008.

[44] A. Barenco, C. H. Bennett, R. Cleve, D. DiVinchenzo, N. Margolus, P. Shor, T. Sleator, J. Smolin, and H. Weinfurter, "Elementary gates for quantum computation," *The American Physical Society*, vol. 52, pp. 3457–3467, 1995.

[45] D. M. Miller, R. Wille, and Z. Sasanian, "Elementary quantum gate realizations for multiple-control Toffolli gates," in *International Symposium on Multiple-Valued Logic*, pp. 217–222, IEEE, 2011.

[46] M. Saeedi, M. S. Zamani, M. Sedighi, and Z. Sasanian, "Synthesis of reversible circuit using cycle-based approach," *J. Emerg. Technol. Comput. Syst.*, vol. 6, no. 4, 2010.

[47] D. M. Miller, D. Maslov, and G. W. Dueck, "A transformation based algorithm for reversible logic synthesis," in *Design Automation Conf.*

[48] P. Kerntopf, "A new heuristic algorithm for reversible logic synthesis," in *Design Automation Conf.*, pp. 834–837, 2004.

[49] P. Gupta, A. Agrawal, and N. K. Jha, "An algorithm for synthesis of reversible logic circuits," *IEEE Trans. on CAD*, vol. 25, no. 11, pp. 2317–2330, 2006.

[50] D. Maslov, G. W. Dueck, and D. M. Miller, "Techniques for the synthesis of reversible Toffoli networks," *ACM Trans. on Design Automation of Electronic Systems*, vol. 12, no. 4, 2007.

[51] K. Fazel, M. Thornton, and J. Rice, "ESOP-based Toffoli gate cascade generation," in *Pacific Rim Conference on Communications, Computers and Signal Processing*, pp. 206–209, 2007.

[52] R. Wille and R. Drechsler, "BDD-based synthesis of reversible logic for large functions," in *Design Automation Conf.*, pp. 270–275, 2009.

[53] I. Wegener, *Branching programs and binary decision diagrams: theory and applications.* Society for Industrial and Applied Mathematics, 2000.

[54] R. Wille and R. Drechsler, "Effect of BDD optimization on synthesis of reversible and quantum logic," *Electronic Notes in Theoretical Computer Science*, vol. 253, no. 6, pp. 57–70, 2010.

[55] M. Soeken, L. Tague, G. W. Dueck, and R. Drechsler, "Ancilla-free synthesis of large reversible functions using binary decision diagrams," *Journal of Symbolic Computation*, 2016.

[56] B. Mealy and F. Tappero, *Free Range VHDL*. freerangefactory.org, 2012.

[57] D. E. Thomas and P. R. Moorby, *The Verilog Hardware Description Language*. Springer, 2002.

[58] S. Sutherland, S. Davidmann, and P. Flake, *System Verilog for Design and Modeling*. Kluwer Academic Publishers, 2004.

[59] R. Wille, J. Stoppe, E. Schönborn, K. Datta, and R. Drechsler, "RevVis: Visualization ofstructures and properties in reversible circuit," in *Reversible Computation*, pp. 111–124, 2014.

[60] R. Wille, M. Soeken, and R. Drechsler, "Reducing the number of lines in reversible circuits," in *Design Automation Conf.*, pp. 647–652, 2010.

[61] R. Wille, S. Offermann, and R. Drechsler, "SyReC: A programming language for synthesis of reversible circuits," in *Forum on Specification and Design Languages*, pp. 184–189, 2010.

[62] R. Wille, M. Soeken, E. Schönborn, and R. Drechsler, "Circuit line minimization in the HDL-based synthesis of reversible logic," in *IEEE Annual Symposium on VLSI*, pp. 213–218, 2012.

[63] E. Schönborn, *Scalable Design and Synthesis of Reversible Circuits*. PhD thesis, University of Bremen, 2016.

[64] T. Yokoyama and R. Glück, "A reversible programming language and its invertible self-interpreter," in *Symp. on Partial evaluation and semantics-based program manipulation*, pp. 144–153, 2007.

[65] M. Chuang and C. Wang, "Synthesis of reversible sequential elements," in *ASP Design Automation Conf.*, pp. 420 –425, 2007.

[66] M. Lukac and M. Perkowski, "Quantum finite state machines as sequential quantum circuits," in *Int'l Symp. on Multi-Valued Logic*, pp. 92–97, 2009.

[67] R. Wille, D. Große, L. Teuber, G. W. Dueck, and R. Drechsler, "RevLib: an online resource for reversible functions and reversible circuits," pp. 220–225, 2008. RevLib is available at http://www.revlib.org.

[68] Y. Takahashi and N. Kunihiro, "A linear-size quantum circuit for addition with no ancillary qubits," *Quantum Information and Computation*, vol. 5, pp. 440–448, 2005.

[69] H. B. Axelsen, "Clean translation of an imperative reversible programming language," in *Int'l Conf. on Compiler Construction*, pp. 144–163, 2011.

[70] M. Nielsen and I. Chuang, *Quantum Computation and Quantum Information*. Cambridge Univ. Press, 2000.

[71] R. Wille, O. Keszöcze, L. Othmer, M. K. Thomsen, and R. Drechsler, "Generating and checking control logic in the hdl-based design of reversible circuits," in *Reversible Computation*, pp. 160–166, 2016.

[72] C. A. R. Hoare, "An axiomatic basis for computer programming," *Communications of the ACM*, vol. 12, no. 10, pp. 576–580, 1969.

[73] K. Datta, V. Shrivastav, I. Sengupta, and H. Rahaman, "Reversible logic implementation of AES algorithm," in *International Conference on Design Technology of Integrated Systems in Nanoscale Era*, pp. 140–144, 2013.